IN THE UNITED STATES DISTRICT COURT
FOR THE DISTRICT OF DELAWARE

TRANSMETA CORPORATION, Plaintiff,	)	C.A. No. 06-633 (GMS)
v.	)	**JURY TRIAL DEMANDED**
INTEL CORPORATION, Defendant.	)	

FIRST AMENDED COMPLAINT

Plaintiff Transmeta Corporation ("Transmeta"), for its complaint against defendant Intel Corporation ("Intel"), alleges:

1. This is an action for patent infringement arising under the patent laws of the United States, Title 35, United States Code. This Court has jurisdiction over this action pursuant to 28 U.S.C. §§ 1331 and 1338(a).

2. Venue within this judicial district is proper under 28 U.S.C. §§ 1391(b) and (c), and 1400(b).

3. Plaintiff Transmeta is a corporation organized and existing under the laws of the State of Delaware, having its principal place of business at 3990 Freedom Circle, Santa Clara, California 95054. Transmeta develops and licenses innovative computing, microprocessor and semiconductor technologies and related intellectual property. Founded in 1995, Transmeta first became known for designing, developing and selling highly efficient x86-compatible software-based microprocessors, which deliver a balance of low power consumption, high performance, low cost and small size suited for diverse computing platforms. Transmeta also develops and licenses advanced power management technologies for increasing power

efficiency and controlling leakage in semiconductor and computing devices. Transmeta also provides, through strategic alliances and under contract, engineering services that leverage its microprocessor design and development capabilities.

4. On information and belief, defendant Intel is a corporation organized and existing under the laws of the State of Delaware, having a principal place of business at 2200 Mission College Blvd., Santa Clara, California 95054. Intel develops, manufactures, markets and sells in the United States and abroad processors used in portable and desktop computers and other electronic devices. Among the processors that Intel has developed, manufactured, marketed and sold, are processors within at least the following product architecture families: P6, Pentium 4, Pentium M, Core and Core 2.

5. U.S. Patent 7,100,061 B2 ("the '061 patent"), entitled "Adaptive Power Control," was duly and legally issued on August 29, 2006. Transmeta is the owner by assignment of all right, title and interest in and to the '061 patent, including the right to sue and recover for past infringements thereof. A copy of the '061 patent is attached as Exhibit A.

6. U.S. Patent 5,895,503 ("the '503 patent"), entitled "Address Translation Method and Mechanism Using Physical Address Information Including During a Segmentation Process," was duly and legally issued on April 20, 1999. Transmeta is the owner by assignment of all right, title and interest in and to the '503 patent, including the right to sue and recover for past infringements thereof. A copy of the '503 patent is attached as Exhibit B.

7. U.S. Patent 6,226,733 B1 ("the '733 patent"), entitled "Address Translation Mechanism and Method in a Computer System," was duly and legally issued on May 1, 2001. Transmeta is the owner by assignment of all right, title and interest in and to the '733

patent, including the right to sue and recover for past infringements thereof. A copy of the '733 patent is attached as Exhibit C.

8. U.S. Patent 6,430,668 B2 ("the '668 patent"), entitled "Speculative Address Translation For Processor Using Segmentation and Optical Paging," was duly and legally issued on August 6, 2002. Transmeta is the owner by assignment of all right, title and interest in and to the '668 patent, including the right to sue and recover for past infringements thereof. A copy of the '668 patent is attached as Exhibit D.

9. U.S. Patent 6,813,699 B1 ("the '699 patent"), entitled "Speculative Address Translation For Processor Using Segmentation And Optional Paging," was duly and legally issued on November 2, 2004. Transmeta is the owner by assignment of all right, title and interest in and to the '699 patent, including the right to sue and recover for past infringements thereof. A copy of the '699 patent is attached as Exhibit E.

10. U.S. Patent 5,493,687 ("the '687 patent"), entitled "RISC Microprocessor Architecture Implementing Multiple Typed Register Sets," was duly and legally issued on February 20, 1996. Transmeta is the owner by assignment of all right, title and interest in and to the '687 patent, including the right to sue and recover for past infringements thereof. A copy of the '687 patent is attached as Exhibit K.

11. U.S. Patent 5,838,986 ("the '986 patent"), entitled "RISC Microprocessor Architecture Implementing Multiple Typed Register Sets," was duly and legally issued on November 17, 1998. Transmeta is the owner by assignment of all right, title and interest in and to the '986 patent, including the right to sue and recover for past infringements thereof. A copy of the '986 patent is attached as Exhibit F.

12. U.S. Patent 6,044,449 ("the '449 patent"), entitled "RISC Microprocessor Architecture Implementing Multiple Typed Register Sets," was duly and legally issued on March 28, 2000. Transmeta is the owner by assignment of all right, title and interest in and to the '449 patent, including the right to sue and recover for past infringements thereof. A copy of the '449 patent is attached as Exhibit G.

13. U.S. Patent 5,737,624 ("the '624 patent"), entitled "Superscalar RISC Instruction Scheduling," was duly and legally issued on April 7, 1998. Transmeta is the owner by assignment of all right, title and interest in and to the '624 patent, including the right to sue and recover for past infringements thereof. A copy of the '624 patent is attached as Exhibit H.

14. U.S. Patent 5,974,526 ("the '526 patent"), entitled "Superscalar RISC Instruction Scheduling," was duly and legally issued on October 26, 1999. Transmeta is the owner by assignment of all right, title and interest in and to the '526 patent, including the right to sue and recover for past infringements thereof. A copy of the '526 patent is attached as Exhibit I.

15. U.S. Patent 6,289,433 B1 ("the '433 patent"), entitled "Superscalar RISC Instruction Scheduling," was duly and legally issued on September 11, 2001. Transmeta is the owner by assignment of all right, title and interest in and to the '433 patent, including the right to sue and recover for past infringements thereof. A copy of the '433 patent is attached as Exhibit J.

16. Defendant Intel has infringed and is continuing to infringe the '061 patent by making, using, selling, offering to sell and/or importing, and by actively inducing and contributing to others' use, sale, offer to sell and/or importing, processors with Enhanced

SpeedStep Technology including at least such processors in the Pentium 4, Pentium M, Core and Core 2 families.

17. Defendant Intel has infringed and is continuing to infringe the ‘503, ‘733, ‘668, and ‘699 patents by making, using, selling, offering to sell and/or importing, and by actively inducing and contributing to others’ use, sale, offer to sell and/or importing, processors including at least processors in the Pentium 4 family.

18. Defendant Intel has infringed and is continuing to infringe the ‘687, ‘986 and ‘449 patents by making, using, selling, offering to sell and/or importing, and by actively inducing and contributing to others’ use, sale, offer to sell and/or importing, processors including at least processors in the P6, Pentium 4, Pentium M, Core and Core 2 families.

19. Defendant Intel has infringed and is continuing to infringe the ‘624, ‘526 and ‘433 patents by making, using, selling, offering to sell and/or importing, and by actively inducing and contributing to others’ use, sale, offer to sell and/or importing, processors including at least processors in the P6, Pentium M, Core and Core 2 families.

20. Transmeta has complied with 35 U.S.C. § 287(a) to the extent applicable and, on information and belief, Intel’s infringements of the aforementioned patents have been willful and deliberate and with full knowledge of the patents.

21. Transmeta has been and continues to be damaged by Intel’s infringements of the aforementioned patents, and will be irreparably injured unless those activities are promptly enjoined by this Court.

22. Transmeta does not have an adequate remedy at law.

WHEREFORE, plaintiff Transmeta Corporation prays for judgment against defendant Intel Corporation as follows:

A. That Transmeta is the owner of the ‘061, ‘503, ‘733, ‘668, ‘699, ‘687, ‘986, ‘449, ‘624, ‘526 and ‘433 patents, and of all rights of recovery thereunder, and that such patents are good and valid in law;

B. That Intel has infringed each of the ‘061, ‘503, ‘733, ‘668, ‘699, ‘687, ‘986, ‘449, ‘624, ‘526 and ‘433 patents;

C. That Transmeta be awarded damages caused by Intel’s infringements, together with pre-judgment and post-judgment interest;

D. That Intel’s infringements be adjudged willful and said damages be trebled pursuant to 35 U.S.C. § 284;

E. Preliminarily and permanently enjoining Intel, its officers, agents, servants, employees, attorneys, all parent and subsidiary corporations and affiliates, their assigns and successors in interest, and those persons in active concert or participation with any of them who receive notice of the injunction, including distributors and customers, enjoining them from continuing acts of infringement of the ‘061, ‘503, ‘733, ‘668, ‘699, ‘687, ‘986, ‘449, ‘624, ‘526 and ‘433 patents;

F. Adjudging this an exceptional case and awarding to Transmeta its reasonable attorneys’ fees pursuant to 35 U.S.C. § 285;

G. Awarding to Transmeta its costs and disbursements incurred in this action; and

H. Awarding to Transmeta such other and further relief as this Court may deem just and proper.

DEMAND FOR JURY TRIAL

Pursuant to Federal Rule of Civil Procedure 38(b), Transmeta demands trial by jury of all issues so triable.

MORRIS, NICHOLS, ARSHT & TUNNELL LLP

/s/ Karen Jacobs Louden

Jack B. Blumenfeld (#1014)
Karen Jacobs Louden (#2881)
klouden@mnat.com
1201 N. Market Street
P.O. Box 1347
Wilmington, DE 19899
(302) 658-9200
Attorneys for plaintiff Transmeta Corporation

OF COUNSEL:

Robert C. Morgan
Laurence S. Rogers
Steven Pepe
ROPES & GRAY LLP
1251 Avenue of the Americas
New York, NY 10020
(212) 596-9000

Norman H. Beamer
ROPES & GRAY LLP
525 University Avenue
Palo Alto, CA 94301
(650) 617-4000

John O'Hara Horsley
TRANSMETA CORPORATION
3990 Freedom Circle
Santa Clara, CA 95054
(408) 919-3000

December 12, 2006
548894

CERTIFICATE OF SERVICE

I, the undersigned, hereby certify that on December 12, 2006, I electronically filed the foregoing with the Clerk of the Court using CM/ECF, which will send notification of such filing(s) to the following:

Josy W. Ingersoll

I also certify that copies were caused to be served on December 12, 20006, upon the following in the manner indicated:

BY HAND

Josy W. Ingersoll
John W. Shaw
YOUNG, CONAWAY, STARGATT & TAYLOR, LLP
The Brandywine Building
1000 West Street, 17th Flr.
Wilmington, DE 19801

BY FEDERAL EXPRESS

Jared Bobrow
WEIL, GOTSHAL & MANGES LLP
201 Redwood Shores Parkway
Redwood Shores, CA 94065

/s/ Karen Jacobs Louden

Karen Jacobs Louden
klouden@mnat.com

EXHIBIT A

US007100061B2

(12) **United States Patent**
Halepete et al.

(10) **Patent No.: US 7,100,061 B2**
(45) **Date of Patent: Aug. 29, 2006**

(54) **ADAPTIVE POWER CONTROL**

(75) Inventors: **Sameer Halepete**, San Jose, CA (US); **H. Peter Anvin**, San Jose, CA (US); **Zongjian Chen**, Palo Alto, CA (US); **Godfrey P. D'Souza**, San Jose, CA (US); **Marc Fleischmann**, Menlo Park, CA (US); **Keith Klayman**, Sunnyvale, CA (US); **Thomas Lawrence**, Mountain View, CA (US); **Andrew Read**, Sunnyvale, CA (US)

(73) Assignee: **Transmeta Corporation**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 90 days.

(21) Appl. No.: **09/484,516**

(22) Filed: **Jan. 18, 2000**

(65) **Prior Publication Data**

US 2002/0116650 A1 Aug. 22, 2002

(51) **Int. Cl.**
G06F 1/30 (2006.01)

(52) **U.S. Cl.** **713/322**

(58) **Field of Classification Search** 713/300–340
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,086,501 A		2/1992	DeLuca et al.	395/550
5,167,024 A		11/1992	Smith et al.	395/375
5,201,059 A		4/1993	Nguyen	395/800
5,204,863 A		4/1993	Saint-Joigny et al.	371/16.3
5,218,704 A		6/1993	Watts, Jr. et al.	395/750
5,222,239 A		6/1993	Rosch	395/750
5,230,055 A		7/1993	Katz et al.	395/750
5,239,652 A		8/1993	Seibert et al.	395/750
5,422,806 A		6/1995	Chen et al.	364/149
5,461,266 A		10/1995	Koreeda et al.	307/125
5,502,838 A		3/1996	Kikinis	395/550
5,511,203 A		4/1996	Wisor et al.	395/750
5,560,020 A		9/1996	Nakatani et al.	395/750
5,572,719 A	*	11/1996	Biesterfeldt	713/500
5,592,173 A		1/1997	Lau et al.	342/357
5,628,001 A	*	5/1997	Cepuran	713/501
5,630,110 A	*	5/1997	Mote, Jr.	713/501
5,682,093 A		10/1997	Kivela	323/273
5,687,114 A	*	11/1997	Khan	365/185.03
5,692,204 A		11/1997	Rawson et al.	395/750
5,710,929 A	*	1/1998	Fung	713/322
5,713,030 A	*	1/1998	Evoy	713/322
5,717,319 A		2/1998	Jokinen	323/280
5,719,800 A		2/1998	Mittal et al.	364/707
5,726,901 A	*	3/1998	Brown	
5,745,375 A	*	4/1998	Reinhardt et al.	700/286
5,752,011 A		5/1998	Thomas et al.	395/556

(Continued)

FOREIGN PATENT DOCUMENTS

EP 0381021 A2 8/1990

(Continued)

OTHER PUBLICATIONS

Andrew S. Tanenbaum, Structured Computer Organization, 1990, Prentice-Hall, Third edition, pp. 11-13.*

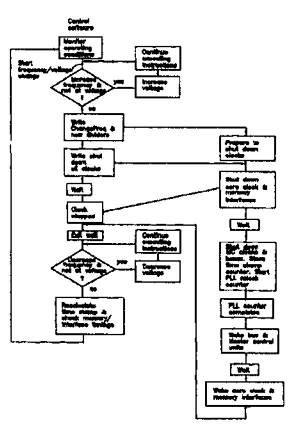
(Continued)

Primary Examiner—Paul R. Myers

(57) **ABSTRACT**

A method for controlling the power used by a computer including the steps of measuring the operating characteristics of a central processor of the computer, determining when the operating characteristics of the central processor are significantly different than required by the operations being conducted, and changing the operating characteristics of the central processor to a level commensurate with the operations being conducted.

59 Claims, 4 Drawing Sheets



U.S. PATENT DOCUMENTS

5,754,869 A		5/1998	Holzhammer et al.	395/750.01
5,757,171 A		5/1998	Babcock	323/271
5,774,703 A	*	6/1998	Weiss et al.	713/501
5,778,237 A		7/1998	Yamamoto et al.	395/750.04
5,781,783 A	*	7/1998	Gunther et al.	713/320
5,812,860 A	*	9/1998	Horden et al.	713/322
5,815,724 A		9/1998	Mates	395/750.04
5,825,674 A		10/1998	Jackson	364/707
5,832,205 A	*	11/1998	Kelly et al.	714/53
5,832,284 A	*	11/1998	Michail et al.	713/322
5,848,281 A		12/1998	Smalley et al.	395/750.04
5,884,049 A		3/1999	Atkinson	395/281
5,894,577 A		4/1999	MacDonald et al.	395/733
5,913,067 A	*	6/1999	Klein	713/300
5,914,996 A	*	6/1999	Huang	327/115
5,919,262 A		7/1999	Kikinis et al.	713/300
5,923,545 A		7/1999	Nguyen	363/24
5,933,649 A		8/1999	Lim et al.	395/750.04
5,940,785 A	*	8/1999	Georgiou et al.	713/322
5,940,786 A		8/1999	Steeby	702/132
5,974,557 A		10/1999	Thomas et al.	713/322
5,996,083 A		11/1999	Gupta et al.	713/322
5,996,084 A		11/1999	Watts	713/323
6,021,500 A	*	2/2000	Wang et al.	713/320
6,047,248 A		4/2000	Georgiou et al.	702/132
6,078,319 A		6/2000	Bril et al.	345/211
6,094,367 A	*	7/2000	Hsu et al.	363/78
6,112,164 A	*	8/2000	Hobson	702/132
6,118,306 A		9/2000	Orton et al.	327/44
6,119,241 A		9/2000	Michail et al.	713/322
6,141,762 A		10/2000	Nicol et al.	713/300
6,157,092 A		12/2000	Hofmann	307/11
6,202,104 B1		3/2001	Ober	710/18
6,216,235 B1		4/2001	Thomas et al.	713/501
6,272,642 B1		8/2001	Pole, III et al.	713/300
6,279,048 B1		8/2001	Fadavi-Ardekani et al.	710/15
6,304,824 B1		10/2001	Bausch et al.	702/64
6,311,287 B1		10/2001	Dischler et al.	713/601
6,314,522 B1		11/2001	Chu et al.	713/322
6,345,363 B1		2/2002	Levy-Kendler	713/320
6,347,379 B1		2/2002	Dai et al.	713/320
6,378,081 B1		4/2002	Hammond	713/501
6,388,432 B1		5/2002	Uchida	323/266
6,415,388 B1		7/2002	Browning et al.	713/322
6,425,086 B1		7/2002	Clark et al.	713/322
6,427,211 B1		7/2002	Watts, Jr.	713/320
6,442,746 B1		8/2002	James et al.	716/14
6,457,135 B1		9/2002	Cooper	713/323
6,477,654 B1		11/2002	Dean et al.	713/300
6,487,668 B1		11/2002	Thomas et al.	713/322
6,510,400 B1		1/2003	Moriyama	702/132
6,510,525 B1		1/2003	Nookala et al.	713/324
6,513,124 B1		1/2003	Furuichi et al.	713/322
6,519,706 B1		2/2003	Ogoro	713/322
6,574,739 B1		6/2003	Kung et al.	713/322
2002/0026597 A1		2/2002	Dai et al.	713/322
2002/0073348 A1		6/2002	Tani	713/300
2002/0083356 A1		6/2002	Dai	713/322
2002/0138778 A1		9/2002	Cole et al.	713/330
2003/0065960 A1		4/2003	Rusu et al.	713/300
2003/0074591 A1		4/2003	McClendon et al.	713/322

FOREIGN PATENT DOCUMENTS

EP	474963 A2	3/1992
EP	0501655 A2	9/1992
JP	409185589 A	7/1997
WO	WO0127728	4/1901

OTHER PUBLICATIONS

Weiser et al.; "Scheduling for Reduced CPU Energy"; Xerox PARC; Palo Alto, CA; Appears in "Proceedings of the First Syymposium on Operating Systems Design and Implementation" USENIX Association; Nov. 1994.

Govil;"Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU"; International Computer Science Institute; Berkeley, CA; Apr. 1995.

Desai et al.; "Sizing of Clock Distribution Networks for High Performance CPU Chips"; Digital Equipment Corp., Hudson, MA; pp. 389-394; 1996.

"High-Speed, Digitally Adustedstepdown Controllers for Notebook CPUS"; Maxim Manual; pp. 11 & 21.

"Operatio U (Refer to Functional Diagram)"; LTC 1736;, Linear Technology Manual; p. 9.

Intel Corporation; "Intel 82801 CAM I/O Controller HUB (ICH3-M)" Datasheet; Jul. 2001.

* cited by examiner

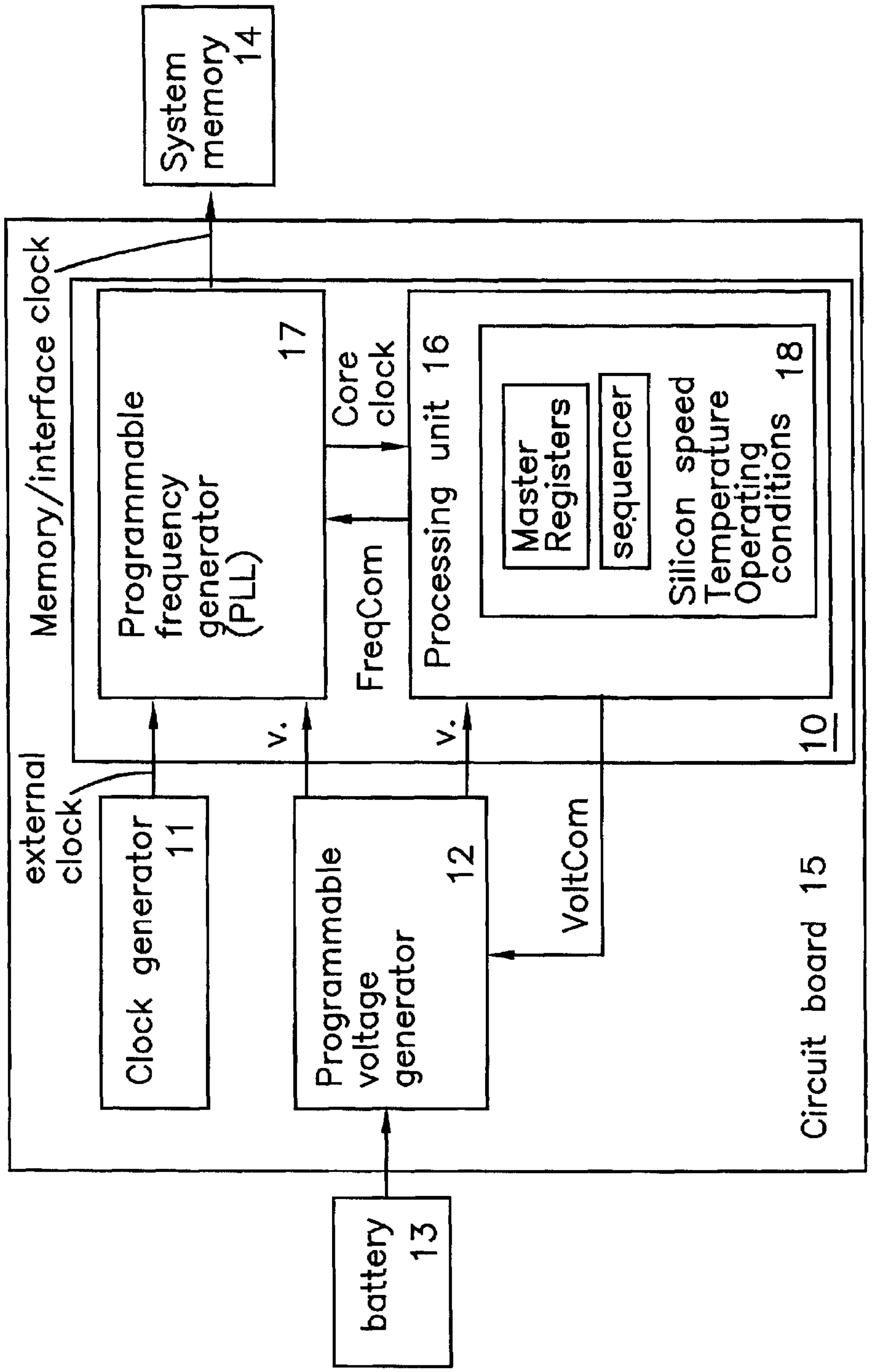

System memory 14
Memory/interface clock
Programmable frequency generator (PLL) 17
Core clock
FreqCom
Processing unit 16
Master Registers
sequencer
Silicon speed
Temperature
Operating conditions 18
v.
v.
10
external clock
Clock generator 11
Programmable voltage generator 12
VoltCom
Circuit board 15
battery 13

Figure 1

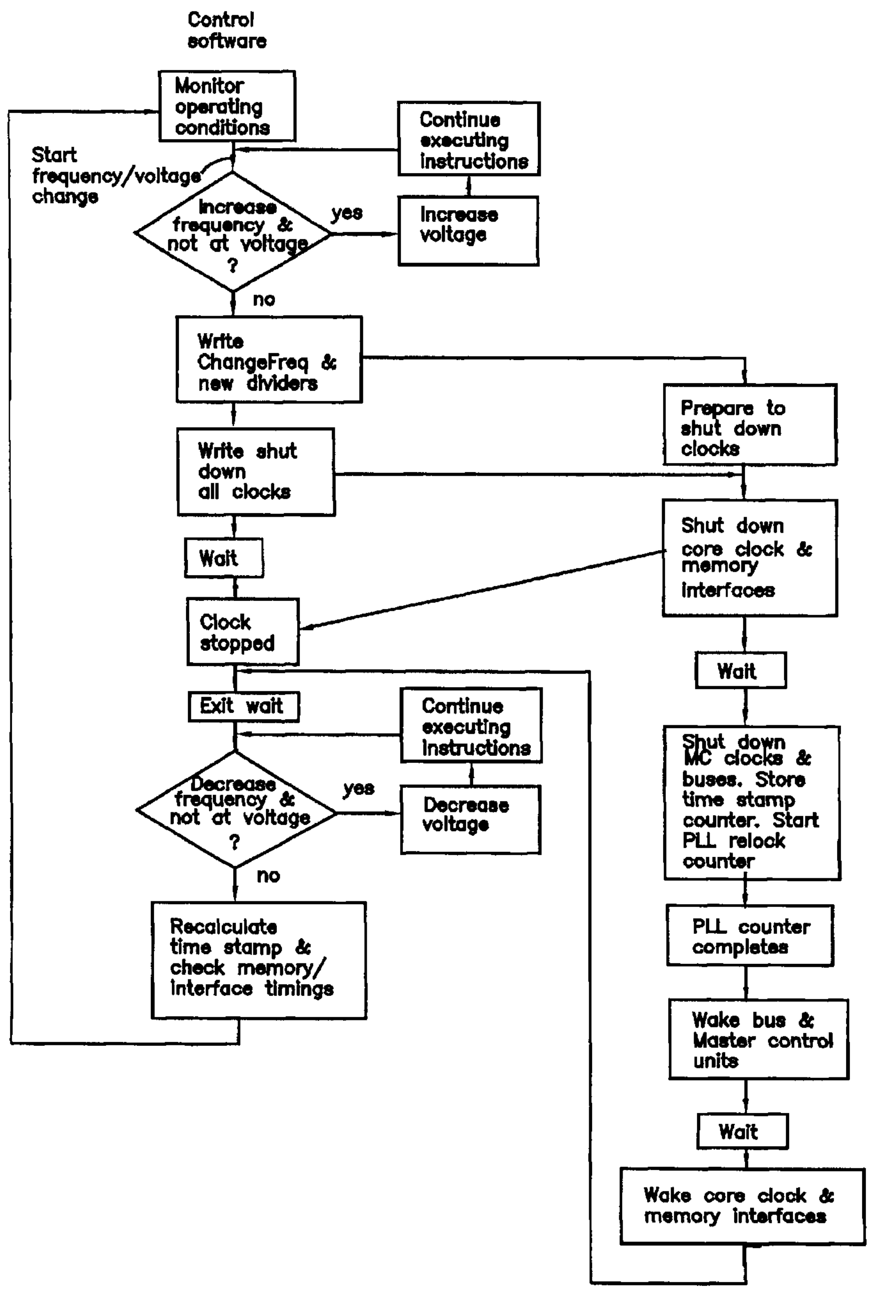

Control software
Monitor operating conditions
Start frequency/voltage change
Continue executing instructions
Increase frequency & not at voltage ?
yes
Increase voltage
no
Write ChangeFreq & new dividers
Prepare to shut down clocks
Write shut down all clocks
Shut down core clock & memory interfaces
Wait
Clock stopped
Wait
Exit wait
Continue executing instructions
Shut down MC clocks & buses. Store time stamp counter. Start PLL relock counter
Decrease frequency & not at voltage ?
yes
Decrease voltage
no
Recalculate time stamp & check memory/ interface timings
PLL counter completes
Wake bus & Master control units
Wait
Wake core clock & memory interfaces

Figure 2

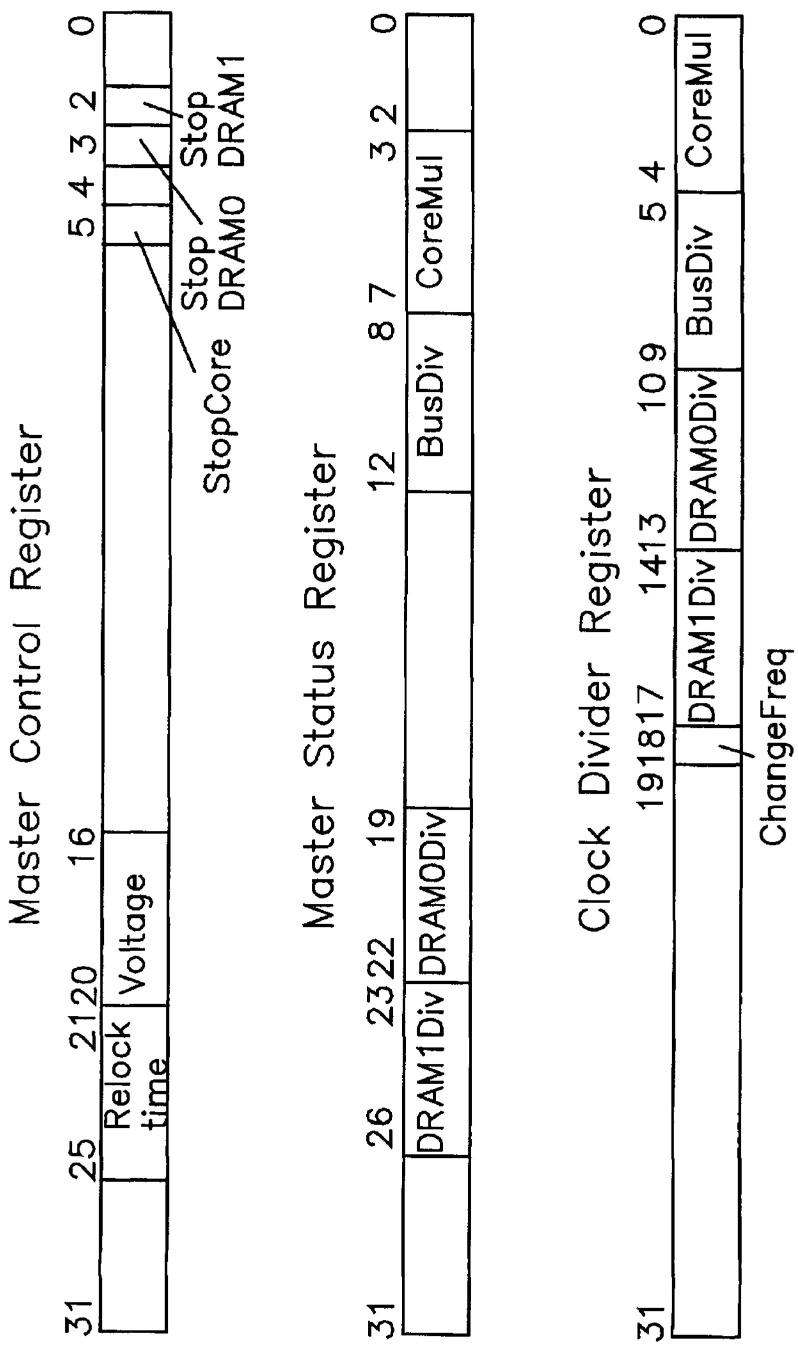

Master Control Register
31
25
21 20
16
5 4 3 2
0
Relock time
Voltage
StopCore
Stop DRAM0
Stop DRAM1
Master Status Register
31
26
23 22
19
12
8 7
3 2
0
DRAM1Div
DRAM0Div
BusDiv
CoreMul
Clock Divider Register
31
19 18 17
14 13
10 9
5 4
0
DRAM1Div
DRAM0Div
BusDiv
CoreMul
ChangeFreq

Figure 3

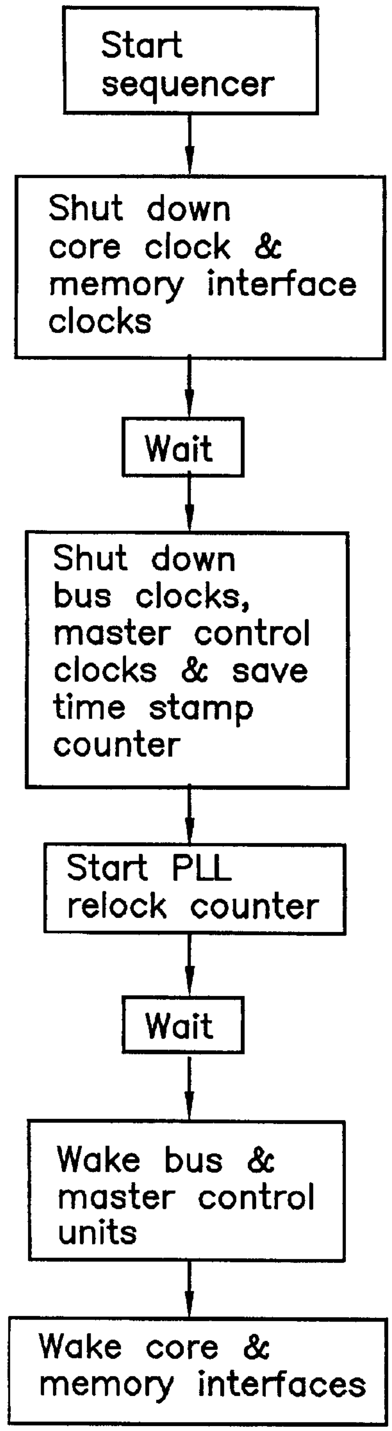

Start sequencer
Shut down core clock & memory interface clocks
Wait
Shut down bus clocks, master control clocks & save time stamp counter
Start PLL relock counter
Wait
Wake bus & master control units
Wake core & memory interfaces

Figure 4

ADAPTIVE POWER CONTROL

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to computer systems and, more particularly, to methods for varying the amount of power used by such systems during use of the systems.

2. History of the Prior Art

A significant problem faced by battery powered computers is the length of time such computers are capable of operating between charges. As computer processors become more capable, they tend to run at faster speeds and dissipate more power. At the same time, the size and weight of portable computers is constantly being reduced to make them more portable. Since batteries tend to be a very significant element of the weight of portable computers and other portable devices, the tendency has been to maintain their size and thus their capacity at a minimum.

A typical portable computer today has an average life of approximately two and one-half hours until its originally-full battery must be recharged.

A great deal of research has been directed to ways for extending the operating life of portable computers. Presently, typical processors include circuitry and software for disabling various power-draining functions of portable computers when those functions are unused for some extensive period. For example, various techniques have been devised for turning off the screen when it has not been used for some selected period. Similar processes measure the length of time between use of hard drives and disable rotation after some period. Another of these processes is adapted to put a central processor into a quiescent condition after some period of inactivity.

In general, these processes are useful in extending the operating life of a portable computer. However, the life still does not extend significantly beyond two and one-half hours for any computer having significant capabilities.

There has been a significant amount of research conducted from which processor requiring less power might be produced. Most processors used in computer systems today are made using CMOS technology. The power consumed by a CMOS integrated circuit is given approximately by $P=CV^2f$, where C is the active switching capacitance, V is the supply voltage, and f is the frequency of operation. The maximum allowable frequency is described by $f_{max}=kV$, where k is a constant.

It is desirable to operate the processor at the lowest possible voltage at a frequency that provides the computing power desired by the user at any given moment. For instance, if the processor is operating at 600 MHz, and the user suddenly runs a compute-intensive process half as demanding, the frequency can be dropped by a factor of two. This means that the voltage can also be dropped by a factor of two. Therefore, power consumption is reduced by a factor of eight. Various methods of implementing this dynamic voltage-frequency scaling have been described in the prior art. All of these involve a component separate from the processor on the system that provides multiple frequencies to multiple system components. Also, they involve state-machines or power-management units on the system to coordinate the voltage-frequency changes. The efficiency of voltage frequency scaling is reduced when the frequency generator is not on the processor. Having a separate power-management unit increases the number of components in the system and the power dissipated by the system. It is also desirable to have the processor control both the voltage it receives and the frequency it receives. As the level of integration increases in processors, they control most of the system clocks; and it is desirable to provide control to the processor to change these clocks so they can be run at just the right frequency. Having a separate clock generator that produces multiple frequencies is not desirable because of the lack of tight coupling.

It is desirable to increase significantly the operating life of portable computers and similar devices.

SUMMARY OF THE INVENTION

It is, therefore, an object of the present invention to increase significantly the operating life of portable computers.

This and other objects of the present invention are realized by a method for controlling the power used by a computer including the steps of utilizing control software to measure the operating characteristics of a processor of the computer, determining when the operating characteristics of the central processor are significantly different than required by the operations being conducted, and changing the operating characteristics of the central processor to a level commensurate with the operations being conducted.

These and other objects and features of the invention will be better understood by reference to the detailed description which follows taken together with the drawings in which like elements are referred to by like designations throughout the several views.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a block diagram of various hardware components of a computer system utilized in accordance with the present invention.

FIG. **2** is a flow chart illustrating the operation of one embodiment of the invention.

FIG. **3** illustrates a number of registers utilized in the hardware components of the system shown in FIG. **1**.

FIG. **4** is a block diagram illustrating the operation of sequencer circuitry which is a part of a processor illustrated in the system of FIG. **1**.

DETAILED DESCRIPTION

FIG. **1** is a block diagram of various hardware components of a computer system utilized in accordance with the present invention to control the operating frequency and voltage of the system. The hardware includes a processor **10**, a clock generator **11**, a programmable voltage generator **12**, system memory (DRAM) **14**, and an external battery (or other power supply) **13**. The processor **10**, clock generator **11**, and voltage generator **12** are all mounted to a circuit board **15** in a manner known to those skilled in the art. The battery **13** and system memory **14** may be electrically connected to the circuit board in a number of possible ways known to those skilled in the art.

The processor **10** includes on the same semiconductor chip a number of components including a processing unit **16** and a programmable frequency generator **17**. The processor **10** also typically includes a number of other components which are known to those skilled in the art but are not pertinent to the present invention and are therefore not illustrated. The processing unit **16** includes a number of logical components including a master control unit **18** which is the central portion for accomplishing clock and voltage control. In the present invention, the master control unit **18**

also includes circuitry for monitoring the operating characteristics of the processor. Various monitoring functions (such as circuitry for accomplishing voltage and frequency monitoring) which are well known to the prior art are included as a part of the logical master control unit **18**. The logical unit **18** may also include circuitry for making available additional information detected by other portions of the computer system in either analogue or digital form (e.g., temperature data). The logical unit **18** also includes circuitry for detecting other operations of the system including commands to be executed from which a particular type of operation to be executed may be determined. A detailed discussion of circuitry for providing various operating characteristics is included in U.S. patent application Ser. No. 09/417,930, entitled *Programmable Event Counter System*, B. Coon et al, filed Oct. 13, 1999, and assigned to the assignee of the present application.

The programmable frequency generator **17** receives an external frequency often referred to as a "slow clock" from the external clock generator **11**. The generator **17** responds to values furnished by control software executing on the processor to produce from the slow clock a core clock for operation of the processing unit **16**, one or more clocks for operation of the various system memory components shown as system memory **14** in the figure, the system bus, and any other components which might utilized a separate clock.

It should be specifically noted that contrasted to prior art systems, the programmable frequency generator is able to provide individual frequencies selectable for each of these components. Thus, prior art arrangements utilize an external clock generator to provide all of the different frequencies utilized by the system. This has a number of effects which are less than desirable. Since the clocks are generated off-chip, the time needed to change frequency is long. Since in an integrated processor all clocks are created from a single slow clock off chip, if the core frequency changes all of the frequencies change with it. Thus, a frequency furnished a single component cannot be changed without affecting a change in other frequencies. The voltage furnished by the external clock generator does not change even though reduced frequencies adapted to provide reduced levels of operations are furnished for various components of the system. A number of other factors slow the response of the system to changes in the various clocks when an external clock is used to generate the various operating frequencies for a system.

The core frequency for the processing unit **16** is generated by multiplying the slow clock by a factor. This factor is computed by the control software of the present invention which monitors the operation of the processor to determine from the characteristics of the processor just what frequency should be selected. The manner in which the monitoring is accomplished and the effect it has on the control of the operating characteristics is described in detail below.

The frequencies at which the other components of the system operate are determined from the core frequency determined by multiplying the slow clock by the core processor factor. For example, a system input/output (I/O) bus typically functions at a much slower frequency than does the processing unit. In the present invention, the control software computes the bus frequency by dividing the core frequency by a value. The process may also be conducted as a table lookup of an already computed value. If the processing unit is conducting its current operations at a normal speed of 400 MHz, a bus frequency of 100 MHz. is derived by dividing the core clock by four. On the other hand, if the processing unit is capable of accomplishing its current operations at a relatively slow speed of 200 MHz, a bus frequency of 100 MHz. is still desirable since bus operations are often the limiting factor in processing operations. In such a case, the control software computes a value of two as the divisor to obtain the bus frequency. It should be noted that although the bus frequency under discussion has been the system I/O bus, the invention may also be used for precisely choosing the operating frequencies for other system buses.

Similarly, various processors are often capable of utilizing system memory having different characteristics one of which is switching speed.

A system may utilize a plurality of interfaces between the processing unit and system memory in order to provide different operating frequencies for system memory which is being utilized. The present invention allows this to be easily accomplished by utilizing different divisors to obtain different values from which the operating frequencies for different system memory units are determined. As will be noted in the following discussion, two different memory frequencies as utilized and more are possible.

Thus, by utilizing the phase-lock-loop generator **17** to determine a core clock frequency and dividing that frequency by a plurality of different values determined by the control software, the operating frequencies for the different components of the system may be individually controlled and furnished to other components of the processor without the necessity of crossing chip boundaries with the consequent slowing caused by negotiating the boundaries.

In order to allow the master control unit **18** to accomplish these operations, the processing unit **16** includes a number of registers which are utilized by the control software and the hardware. These include a master control register **20**, a master status register **21**, and a master clock divider register **22** which are illustrated in FIG. **3**.

Of these registers, the clock divider register **22** stores, among other things, the multiplier computed by the control software for generating the core frequency, the value used as a divisor to obtain the bus frequency from the core frequency, a value used as a divisor to obtain a first system memory frequency from the core frequency, and a value used as a divisor to obtain a second system memory frequency from the core frequency. In addition, the clock divider register **22** stores values used for various other including an indication that a frequency change command has been received.

The master control register includes values pertinent to the present description including the voltage which is to be furnished to the processor as a part of the change of frequency. This register also stores a value indicating the time period allowed for accomplishing the phase-lock-loop relock operation. The master status register also stores the various values used as dividers and the value used as a multiplier to obtain the core frequency along with other significant information.

The various values stored in these registers are utilized, among other things, to control the operations of sequencer circuitry (illustrated in FIG. **4**) which carries out the operations necessary to changing the frequency at which the components of the system operate. The sequencer circuitry carries out the series of steps required by which the phase-lock-loop circuitry is brought to the new frequency and relocked after the processor clock has been shut off.

The operations carried out by the sequencer commence at an idle state which represents the normal condition of the sequencer in the absence of a frequency change operation. When the change frequency command and values are received, the sequencer steps from the idle condition to first

shut down the core clock and the clocks to the various memory interfaces. The sequencer then waits a few cycles before shutting down the bus clock, the master control clock, and saving information sufficient to assure that timing during and after the sequencing is correct. After this delay, the sequencer starts a counter to time the phase-lock-loop relock process. When this count is complete, the sequencer wakes the bus and the master control units. Finally, the sequencer wakes up the core and memory interfaces and awaits another frequency changing operation. The relation of the sequencer to the control software will be described in detail in the discussion of the process of the control software which follows.

FIG. 2 is a flow chart representing the process carried out by one embodiment of the invention. In the figure, the steps described in the left column represent operations accomplished by the control software, while the steps described in the right column represent operations accomplished by the cooperating hardware.

In a first step, the control software monitors various conditions of the processor which relate to power expenditure by the processor. These conditions may include any of those described above including the present frequency and voltage of operation, the temperature of operation, the amount of time the processor spends in one of what may be a number of idle states in which various components of the system are quiescent. For example, if the processor is running in what might be termed its normal mode of operation at a core frequency of 400 MHz. and a voltage of 1.3 volts, the control software may be monitoring the amount of time the processor spends in the "halt" state, the amount of time the processor spends in the "deep sleep" state, and the temperature of the processor. The deep sleep state is a state in which power is furnished only to the processor and to DRAM memory. In this state, the processor are all off and it does not respond to any interrupts. The halt state is a state in which the core clock has been stopped but the processor responds to most interrupts. If the processor is spending more than a preselected increment of its operation in these states while operating at normal frequency and voltage, then power is being wasted. The detection of such operating characteristics therefore may indicate that the frequency and voltage of operation should be reduced.

On the other hand, it may be found that the processor is functioning at a reduced frequency and voltage and that a series commands have been furnished to be executed by the processor which require greater processing power. In such a case, these characteristics suggest that it may be desirable to increase the voltage and frequency of operation in order to handle these commands.

Consequently, the control software detects operating characteristics and determines whether those characteristics indicate that the frequency and voltage of operation should be changed. From the possible sets of conditions, the control software detects the particular set involved and computes correct values for the core clock frequency, the core clock frequency multiplier, the various DRAM clock frequency dividers, and the bus frequency divider. If any other components of the circuitry receive their own clocks, then multipliers or dividers for these values are computed. It should be noted that the control software may actually compute the various values required for the given characteristics which have been determined or may utilize a lookup table storing precomputed values.

At a next step, the software reviews the values computed and determines whether the frequency is to be increased. If the frequency is to be increased, it is first necessary that the voltage be increased to allow the processor to function at a higher frequency. In such a case, it is first necessary to increase the voltage level of operation. The typical power supplies offer a number of pins (often five) by which different operating voltages may be selected. This allows a range of different voltages to be provided. Consequently, the control software simply furnishes a correct value on the input pins of the power supply to cause the computed voltage to be furnished to the frequency generator and to the processor. In one embodiment, the voltage increase is accomplished by providing a level to be reached and a time period for the voltage to settle to this level.

It should be noted that the voltage may be increased in a single step, an action which would typically cause phase-locked-loop circuitry of a frequency generator to lose its lock and would create a large surge of current causing the currently-available voltage regulator circuitry to initiate a system reset. This problem may be eliminated with future voltage regulator circuitry. Alternatively, the voltage may be increased in a series of small steps which would not have this effect. For example, if increases of approximately 50 millivolts are enabled, then the frequency generator will remain stable during the voltage increase and a system reset will not occur. This offers the advantage that the processor may continue to execute commands during the period in which the voltage change is taking place.

If the control software was not increasing but rather decreasing frequency of operation at the previous step, then the original voltage level is not changed at this time. In either case, the control software then goes through a sequence of steps in which various operations of the processor are prepared for shutdown so that the system clocks can be changed. With a particular processor such as that referred to in the patent application described above, this includes flushing a gated store buffer, suspending bus and direct memory access (DMA) operations, and enabling self-refreshing circuitry for system DRAM memory.

With these processor operations shut down, the control software transfers the new divider values and writes a bit indicating a frequency change is to occur. The hardware stores the divider values in the clock divider register and the change frequency indicator. This starts the hardware process of the sequencer. The control software then writes "stop core," "stop DRAM0" and "stop DRAM1" bits of the master command register to stop the clocks being furnished to these components.

Writing the master control register bits to stop the clock frequencies and the values to the hardware causes the hardware to commence the remainder of the frequency changing operation utilizing the sequencer circuitry described above. At this point, the software effectively goes into a wait state which continues until the core clock is enabled at the new frequency. The sequencer responds to the command by shutting down the core clock and the DRAM memory interfaces. The sequencer pauses for sufficient time to assure that this has happened and then shuts down the bus and master control clocks.

Because the core clock has been stopped, timing must be accomplished based on the external clock furnished to the system during this period. Counter circuitry dependent only on phase-lock-loop relock time is utilized to measure the time allowed for the phase-lock-loop circuitry to lock to the new frequency. At this point, the sequencer utilizes the new values furnished to effect a new value for the core (and

other) frequency. After a safe lock period has passed ("relock time" stored in the master control register), the sequencer wakes the bus and master control units. The sequencer waits a few clocks of the slow frequency and then turns on the core clock and the DRAM interfaces.

Because the internal clocks of the system are shut down during the operation of the sequencer, it is necessary that the system provide a means of maintaining timing consistent with the normal world clock. Computer systems utilize a time stamp counter to keep track of world clock values. The value kept in this counter is utilized for certain operations conducted by the central processing unit. Once the phase-lock-loop circuitry of the frequency generator **17** has been stopped, the value in the time clock counter no longer represents accurate world time. Moreover, when the new frequency is reached and locks in, the rate at which the counter is iterated will change. To provide for accurate time stamp readings, a number of lower-valued bits indicating the last time of program execution held by the time stamp counter are stored. These are furnished to the control software along with the relock time value and the new frequency once the frequencies have restabilized to allow accurate computation of the normal world time.

Once the clocks have been turned on at the new frequencies, the control software ends its wait state and determines whether the operation was to decrease the frequency. Assuming the operation was to increase the frequency, the software then recalculates the time stamp counter value and checks the various interface timings to assure that they are correct. If the operation was to decrease the frequency, the control software causes the voltage to be lowered to the calculated value (either in one or a series of incremental steps) and then recalculates the value for the time stamp counter and checks the interface timings. At that point, the control software begins again to monitor the various conditions controlling the frequency and voltage of operation.

It should be noted that at some point during the monitoring operation it may be found that the processor is functioning at a normal frequency and voltage, that the temperature of operation is below some preselected value, and that a series of processor-intensive commands have been furnished to be executed by the processor. In such a case, these characteristics suggest that it may be desirable to increase the voltage and frequency of operation in order to handle these commands for a period less than would raise operating temperatures beyond a safe level. In such a case, the control software may compute higher frequency and voltage values and a temperature (or a time within which temperature will not increase beyond a selected level) in order to cause the hardware to move to this higher frequency state of operation. In such a case, the processor executing the process illustrated effectively ramps up the frequency and voltage so that the processor "sprints" for a short time to accomplish the desired operations. This has the effect of allowing a processor which nominally runs at a lower frequency to attain operational rates reached by more powerful processors during those times when such rates are advantageous.

Although the present invention has been described in terms of a preferred embodiment, it will be appreciated that various modifications and alterations might be made by those skilled in the art without departing from the spirit and scope of the invention. The invention should therefore be measured in terms of the claims which follow.

What is claimed is:

1. A method for controlling power consumption of a computer processor on a chip comprising the steps of:

determining a maximum allowable power consumption level from an operating condition of the processor,

said computer processor determining a maximum frequency which provides power not greater than the allowable power consumption level,

said computer processor determining a minimum voltage which allows operation at the maximum frequency determined, and

dynamically changing the power consumption of the processor by changing frequency and voltage, respectively, to the maximum frequency and the minimum voltage determined, wherein said dynamically changing the power consumption comprises executing instructions in said computer processor while changing voltage at which said computer processor is operated.

2. The method of claim **1**, wherein said dynamically changing the power consumption comprises increasing voltage prior to increasing frequency.

3. The method of claim **1**, wherein said dynamically changing the power consumption comprises lowering frequency prior to lowering voltage.

4. The method of claim **3**, wherein dynamically changing the power consumption further comprises:

executing instructions in said computer processor while lowering voltage at which said computer processor is operated.

5. The method of claim **1**, wherein said dynamically changing the power consumption comprises:

executing said instructions in said computer processor while lowering voltage at which said computer processor is operated.

6. The method of claim **1**, wherein said dynamically changing the power consumption comprises concurrently generating a plurality of frequencies.

7. The method of claim **1**, wherein said operating condition of the processor is internal to the processor.

8. A computing device comprising:

a power supply furnishing selectable output voltages,

a clock frequency source,

a central processor including:

a processing unit for providing values indicative of operating conditions of the central processor, and

a clock frequency generator receiving a clock frequency from the clock frequency source and providing one of a plurality of selectable output clock frequencies to the processing unit;

means for detecting the values indicative of operating conditions of the central processor and causing the power supply and clock frequency generator to furnish an output clock frequency and voltage level for the central processor and to generate concurrently frequencies which are selected for optimum operation of a plurality of functional units of the computing device; and

means for executing instructions in said central processor while changing voltage at which said central processor is operated.

9. A computing device as claimed in claim **8** in which the means for detecting the values indicative of operating conditions of the central processor comprises control software for determining an output clock frequency and voltage level for the central processor adapted to conserve power while maintaining an effective execution rate.

10. A method for controlling the power used by a computer comprising the steps of:

utilizing control software dedicated to a central processor to measure the operating characteristics of the central processor of the computer,

determining when the operating characteristics of the central processor are significantly different than required by the operations being conducted, and

changing the operating characteristics of the central processor to a level commensurate with the operations being conducted in which:

the step of determining when the operating characteristics of the central processor are significantly different than required by the operations being conducted comprising utilizing the control software to determine desirable voltages and frequencies for the operation of the central processor based on the measured operating characteristics, and

the step of changing the operating characteristics of the central processor to a level commensurate with the operations being conducted comprises:

providing signals:

for controlling voltages furnished by a programmable power supply to the central processor,

for controlling frequencies furnished by the central processor to the central processor, and

providing signals for controlling frequencies furnished by the central processor to other functional units of the computer; and

executing instructions in said central processor while changing voltage at which said central processor is operated.

11. A computer comprising:

a power supply furnishing selectable output voltages,

a clock frequency source,

a bus,

system memory,

a central processor including:

a processing unit for providing values indicative of operating conditions of the central processor, and

a clock frequency generator receiving a clock frequency from the clock frequency source and providing a plurality of selectable output clock frequencies to the processing unit; and

means for detecting the values indicative of operating conditions of the central processor and causing the power supply and clock frequency generator to furnish an output clock frequency and voltage level for the central processor and to generate concurrently frequencies which are selected for optimum operation of a plurality of functional units of the computing device including system memory, wherein the means for detecting the values indicative of operating conditions of the central processor is further for causing execution of instructions in said central processor while changing voltage at which said central processor is operated.

12. A computer as claimed in claim 11 in which the means for detecting the values indicative of operating conditions of the central processor comprises control software for determining an output clock frequency and voltage level for the central processor adapted to conserve power while maintaining an effective execution rate.

13. A computing device as claimed in claim 11 in which the means for detecting the values indicative of operating conditions of the central processor causes the clock frequency generator to generate frequencies with are selected for optimum operation of system memory.

14. A computing device as claimed in claim 11 in which the means for detecting the values indicative of operating conditions of the central processor causes the clock frequency generator to generate frequencies which are selected for optimum operation of the bus.

15. A method of controlling a computer processor, comprising:

monitoring operating conditions internal to said computer processor;

determining a frequency and a voltage at which to operate said computer processor, based on said internal operating conditions; and

implementing the determined frequency and voltage, wherein said implementing comprises:

executing instructions in said computer processor while changing voltage at which said computer processor is operated.

16. The method of claim 15 wherein said monitoring comprises said computer processor monitoring operating conditions internal to said computer processor.

17. The method of claim 15, wherein said implementing comprises lowering frequency at which said computer processor is operated.

18. The method of claim 17, wherein said changing voltage comprises lowering voltage at which said computer processor is operated.

19. The method of claim 18, wherein said lowering voltage occurs after said lowering frequency.

20. The method of claim 15, wherein said implementing comprises:

increasing frequency at which said computer processor is operated.

21. The method of claim 20, wherein said changing voltage comprises increasing voltage at which said computer processor is operated.

22. The method of claim 21, wherein said increasing voltage occurs prior to said increasing frequency.

23. A method of controlling a computer processor, comprising:

monitoring idle time of said computer processor;

said computer processor determining a frequency and a voltage at which to operate said computer processor, based on said idle time; and

implementing the determined frequency and voltage, wherein said implementing comprises executing instructions in said computer processor while changing voltage at which said computer processor is operated.

24. The method of claim 23, wherein said implementing comprises:

lowering frequency at which said computer processor is operated prior to lowering voltage at which said computer processor is operated.

25. The method of claim 24, wherein said implementing further comprises:

increasing voltage at which said computer processor is operated prior to increasing frequency at which said computer processor is operated.

26. The method of claim 24, wherein said implementing further comprises:

executing instructions in said computer processor while lowering voltage at which said computer processor is operated.

27. The method of claim 23, wherein said implementing comprises:

increasing voltage at which said computer processor is operated prior to increasing frequency at which said computer processor is operated.

28. The method of claim 23, wherein said monitoring idle time comprises monitoring internal data of said computer processor.

29. The method of claim 23, wherein said implementing comprises:

executing instructions in said computer processor while lowering voltage at which said computer processor is operated.

30. A method of controlling a computer processor, comprising:

monitoring a state of said computer processor;

said computer processor determining a frequency and a voltage at which to operate said computer processor, based on said state; and

implementing the determined frequency and voltage, wherein said implementing comprises executing instructions in said computer processor while changing voltage at which said computer processor is operated.

31. The method of claim 30, wherein said state comprises a sleep state.

32. The method of claim 31, wherein said monitoring further comprises monitoring a halt state of said computer processor.

33. The method of claim 30, wherein said state comprises a halt state.

34. The method of claim 30, wherein said implementing comprises:

lowering frequency at which said computer processor is operated prior to lowering voltage at which said computer processor is operated.

35. The method of claim 34, wherein said implementing further comprises:

increasing voltage at which said computer processor is operated prior to increasing frequency at which said computer processor is operated.

36. The method of claim 34, wherein said implementing further comprises:

executing instructions in said computer processor while lowering voltage at which said computer processor is operated.

37. The method of claim 30, wherein said implementing comprises:

increasing voltage at which said computer processor is operated prior to increasing frequency at which said computer processor is operated.

38. The method of claim 30, wherein said implementing comprises:

executing instructions in said computer processor while lowering voltage at which said computer processor is operated.

39. A method of managing power consumption comprising:

monitoring internal conditions of a computer processor;

based on said internal conditions, determining an allowable power consumption level;

a computer processor determining a voltage-frequency pair for said allowable power consumption level; and

dynamically changing power consumption of the computer processor by implementing said voltage-frequency pair, wherein said dynamically changing power consumption comprises changing voltage at which said computer processor is operated while executing instructions in said computer processor.

40. The method of claim 39, wherein said dynamically changing power consumption comprises:

lowering frequency at which said computer processor is operated prior to lowering voltage at which said computer processor is operated.

41. The method of claim 40, wherein said dynamically changing power consumption further comprises:

increasing voltage at which said computer processor is operated prior to increasing frequency at which said computer processor is operated.

42. The method of claim 40, wherein said implementing further comprises:

executing instructions in said computer processor while lowering voltage at which said computer processor is operated.

43. The method of claim 39, wherein said dynamically changing power consumption comprises:

increasing voltage at which said computer processor is operated prior to increasing frequency at which said computer processor is operated.

44. The method of claim 39, wherein said monitoring comprises monitoring a state of said computer processor.

45. The method of claim 44, wherein said state comprises a halt state.

46. The method of claim 44, wherein said state comprises a sleep state.

47. The method of claim 46, wherein said monitoring further comprises monitoring a halt state of said computer processor.

48. The method of claim 39, wherein said monitoring comprises monitoring a temperature.

49. The method of claim 48, wherein said monitoring further comprises monitoring a state of said computer processor.

50. The method of claim 49, wherein said state comprises a halt state.

51. The method of claim 49, wherein said state comprises a sleep state.

52. The method of claim 51, wherein said monitoring further comprises monitoring a halt state of said computer processor.

53. The method of claim 48, wherein said dynamically changing the power consumption comprises lowering frequency prior to lowering voltage.

54. The method of claim 39, wherein said determining a voltage-frequency pair comprises accessing a table of predetermined voltage-frequency pairs.

55. The method of claim 39, wherein said determining a voltage-frequency pair comprises calculating a voltage-frequency pair.

56. A computing device comprising:

a power supply furnishing selectable output voltages;

a clock frequency source; and

a central processor comprising:

a clock frequency generator receiving a clock frequency from the clock frequency source; and

a processing unit operable to provide values indicative of operating conditions of the central processor and to cause the power supply and the clock frequency generator to furnish a voltage level and an output clock frequency for the central processor, wherein said processing unit is further operable to cause the power supply to cause voltage furnished to the central processor to change while the central processor is executing instructions.

57. The computing device of claim **56**, wherein:

said clock frequency generator is operable to provide one of a plurality of selectable output clock frequencies to the processing unit.

58. The computing device of claim **57**, wherein:

said clock frequency generator is further operable to concurrently generate frequencies for a plurality of functional units of the computing device.

59. The computing device of claim **56**, wherein:

said clock frequency generator is operable to concurrently generate frequencies for a plurality of functional units of the computing device.

* * * * *

EXHIBIT B

US005895503A

United States Patent [19]

Belgard

[11] **Patent Number: 5,895,503**

[45] **Date of Patent: Apr. 20, 1999**

[54] **ADDRESS TRANSLATION METHOD AND MECHANISM USING PHYSICAL ADDRESS INFORMATION INCLUDING DURING A SEGMENTATION PROCESS**

[76] Inventor: **Richard A. Belgard**, 21250 Glenmont Dr., Saratoga, Calif. 95070

[21] Appl. No.: **08/458,479**

[22] Filed: **Jun. 2, 1995**

[51] **Int. Cl.**[6] **G06F 12/10**; G06F 12/06

[52] **U.S. Cl.** **711/202**; 711/200; 711/201; 711/203; 711/206; 711/204; 711/208

[58] **Field of Search** 395/403, 416, 395/413, 418, 412, 419, 410, 414, 415, 421.1

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,084,225	4/1978	Anderson et al.	395/403
5,321,826	6/1994	Crawford et al.	395/416
5,321,836	6/1994	Crawford et al.	395/416
5,335,333	8/1994	Hinton et al.	395/417
5,423,014	6/1995	Hinton et al.	395/403

OTHER PUBLICATIONS

Intel Microprocessors, vol. 1, Intel Corporation, 1993, pp. 2–229 to 2–287.

Computer Architecture A Quantitative Approach, Hennessey and Patterson, pp. 432–497.

U5S 486 Green CPU, United Microelectronics Corporation, 1994–95, pp. 3–1 to 3–26.

The Multics System, Elliott Organick, 1972, pp. 6–7, 38–51.

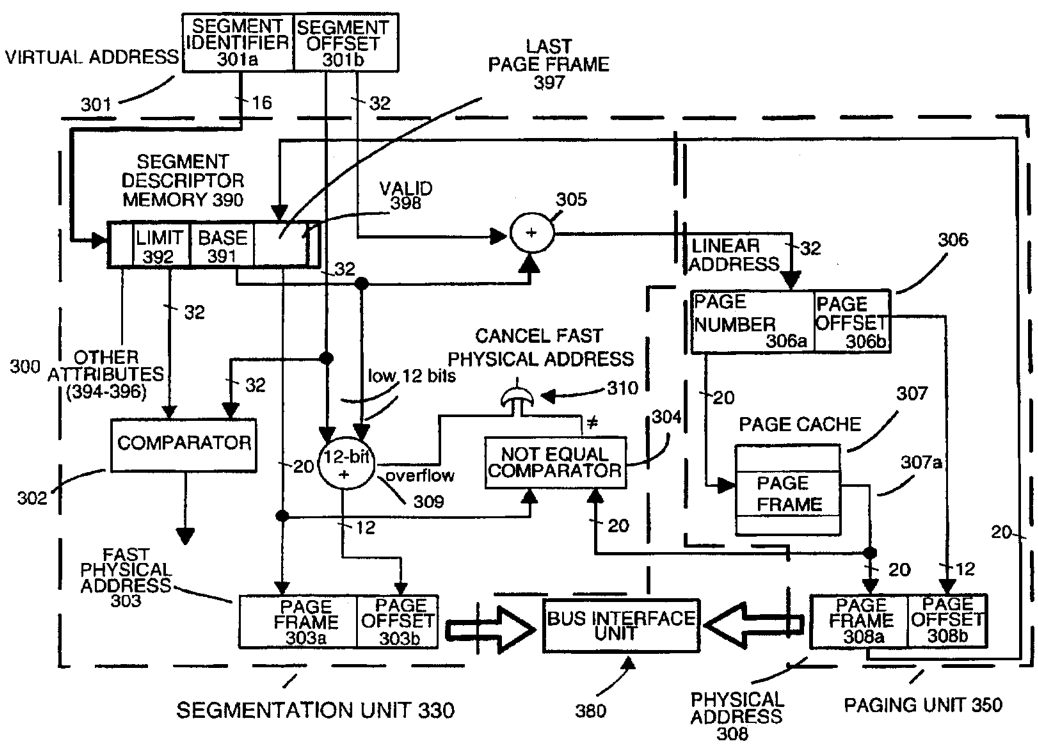
DPS–8 Assembly Instructions, Honeywell Corporation, Apr., 1980, Chapters 3 and 5.

Primary Examiner—Robert W. Downs
Assistant Examiner—Than V. Nguyen
Attorney, Agent, or Firm—Law +

[57] **ABSTRACT**

An improved address translation method and mechanism for memory management in a computer system is disclosed. A segmentation mechanism employing segment registers maps virtual addresses into a linear address space. A paging mechanism optionally maps linear addresses into physical or real addresses. Independent protection of address spaces is provided at each level. Information about the state of real memory pages is kept in segment registers or a segment register cache potentially enabling real memory access to occur simultaneously with address calculation, thereby increasing performance of the computer system.

23 Claims, 5 Drawing Sheets



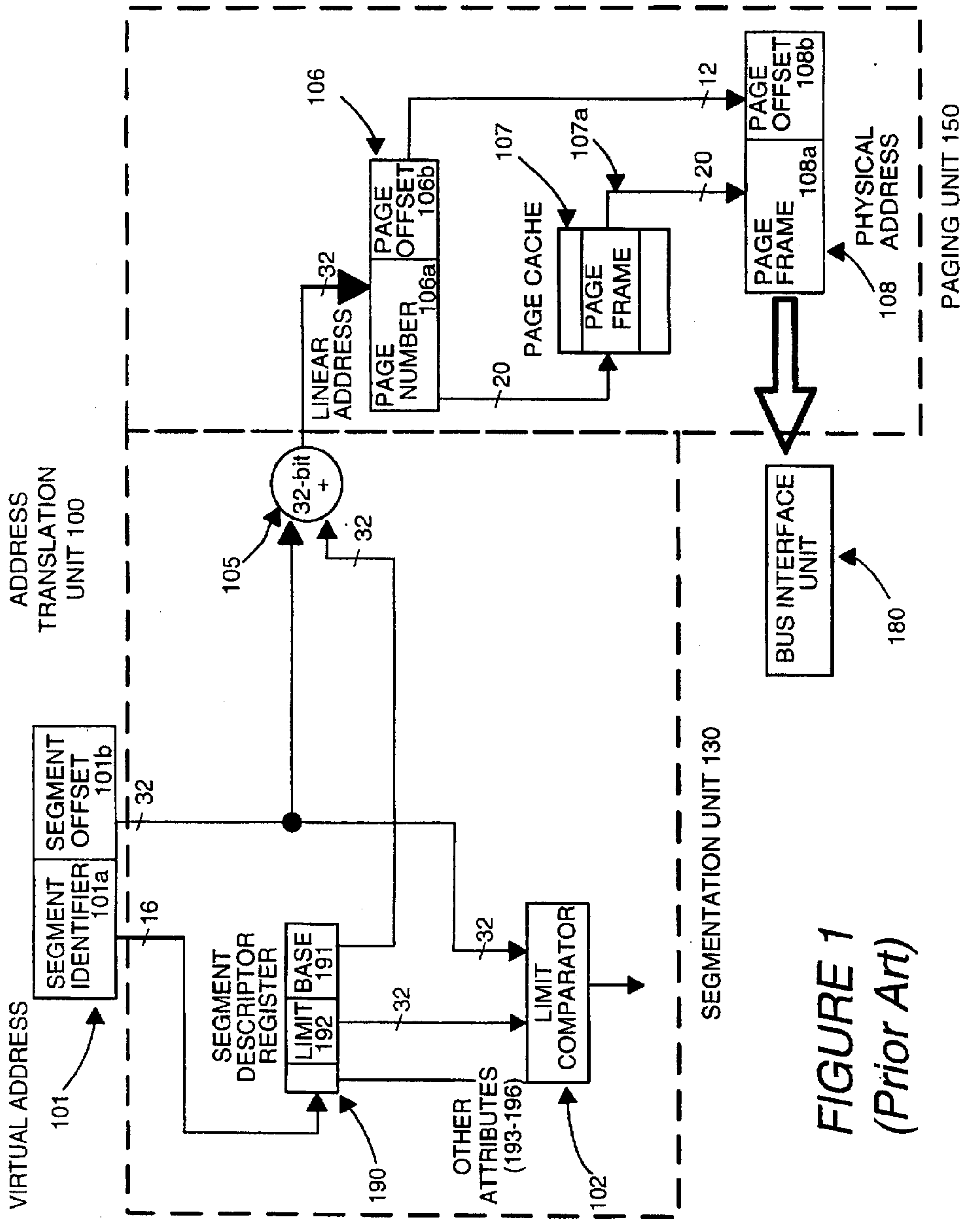
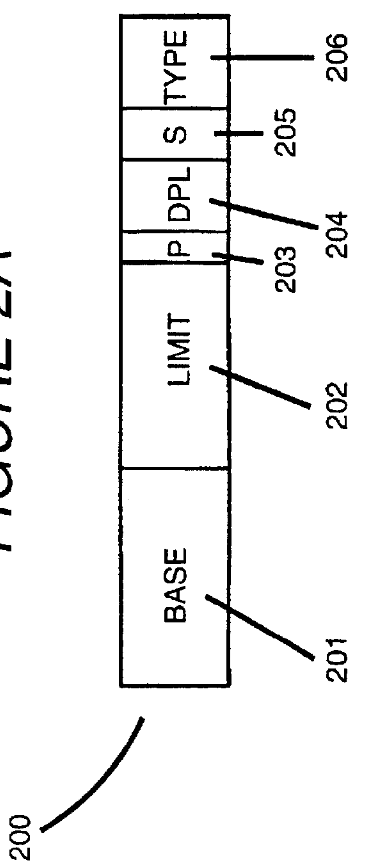


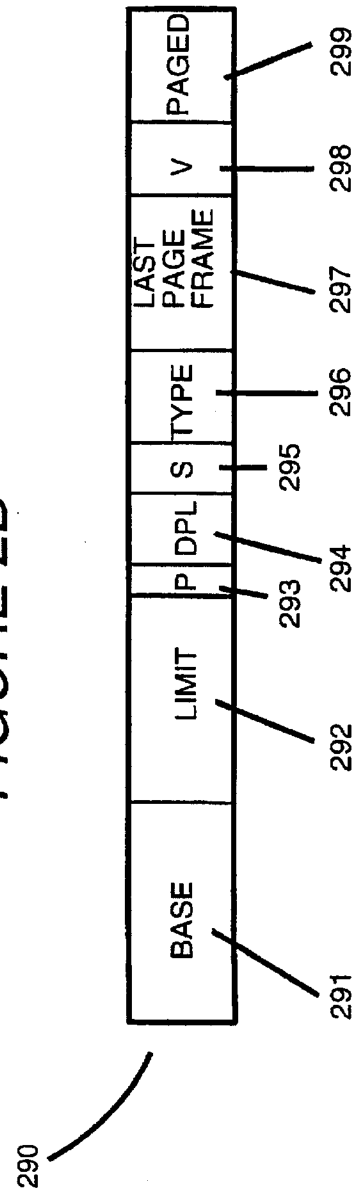
FIGURE 1
(Prior Art)

FIGURE 2A



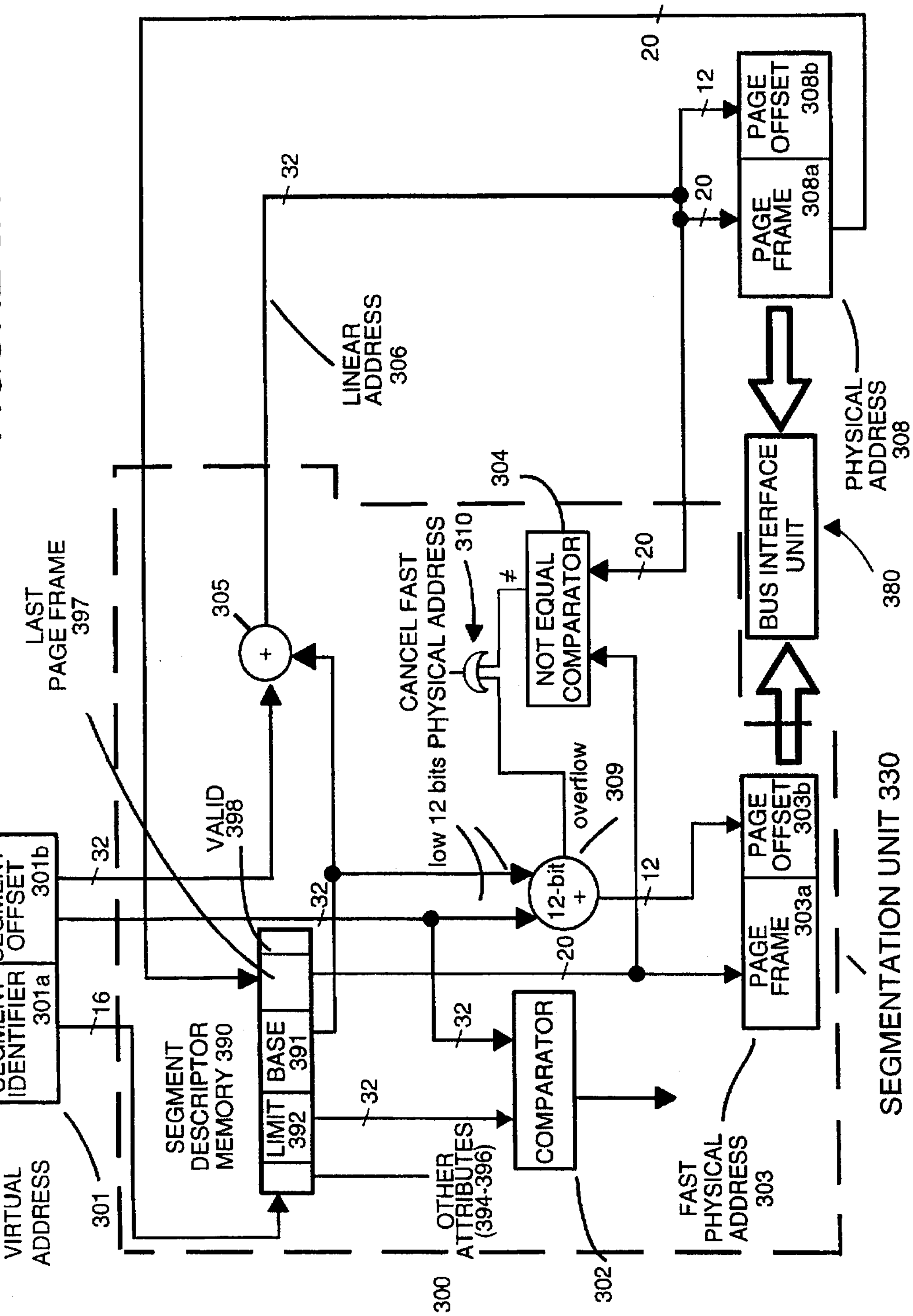
PRIOR ART SEGMENT DESCRIPTOR REGISTER

FIGURE 2B



SEGMENT DESCRIPTOR MEMORY

FIGURE 3A


VIRTUAL ADDRESS
301
SEGMENT IDENTIFIER 301a
SEGMENT OFFSET 301b
16
32
SEGMENT DESCRIPTOR MEMORY 390
LIMIT 392
BASE 391
LAST PAGE FRAME 397
VALID 398
305
+
32
OTHER ATTRIBUTES (394-396)
COMPARATOR
302
300
low 12 bits PHYSICAL ADDRESS
CANCEL FAST PHYSICAL ADDRESS
310
12-bit +
overflow
309
20
12
NOT EQUAL COMPARATOR
≠
304
LINEAR ADDRESS 306
PAGE FRAME 303a
PAGE OFFSET 303b
FAST PHYSICAL ADDRESS 303
BUS INTERFACE UNIT
380
PAGE FRAME 308a
PAGE OFFSET 308b
PHYSICAL ADDRESS 308
SEGMENTATION UNIT 330

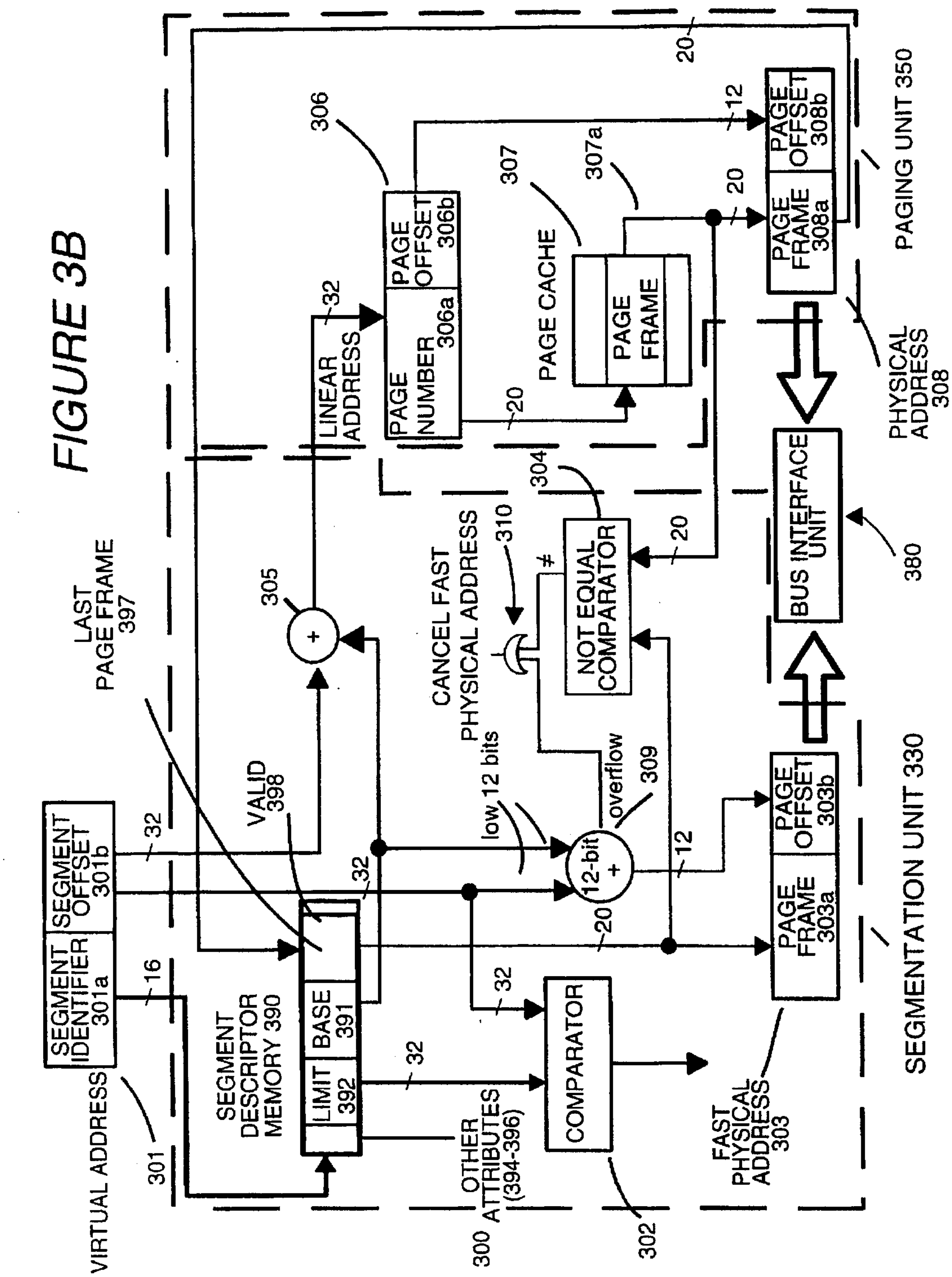
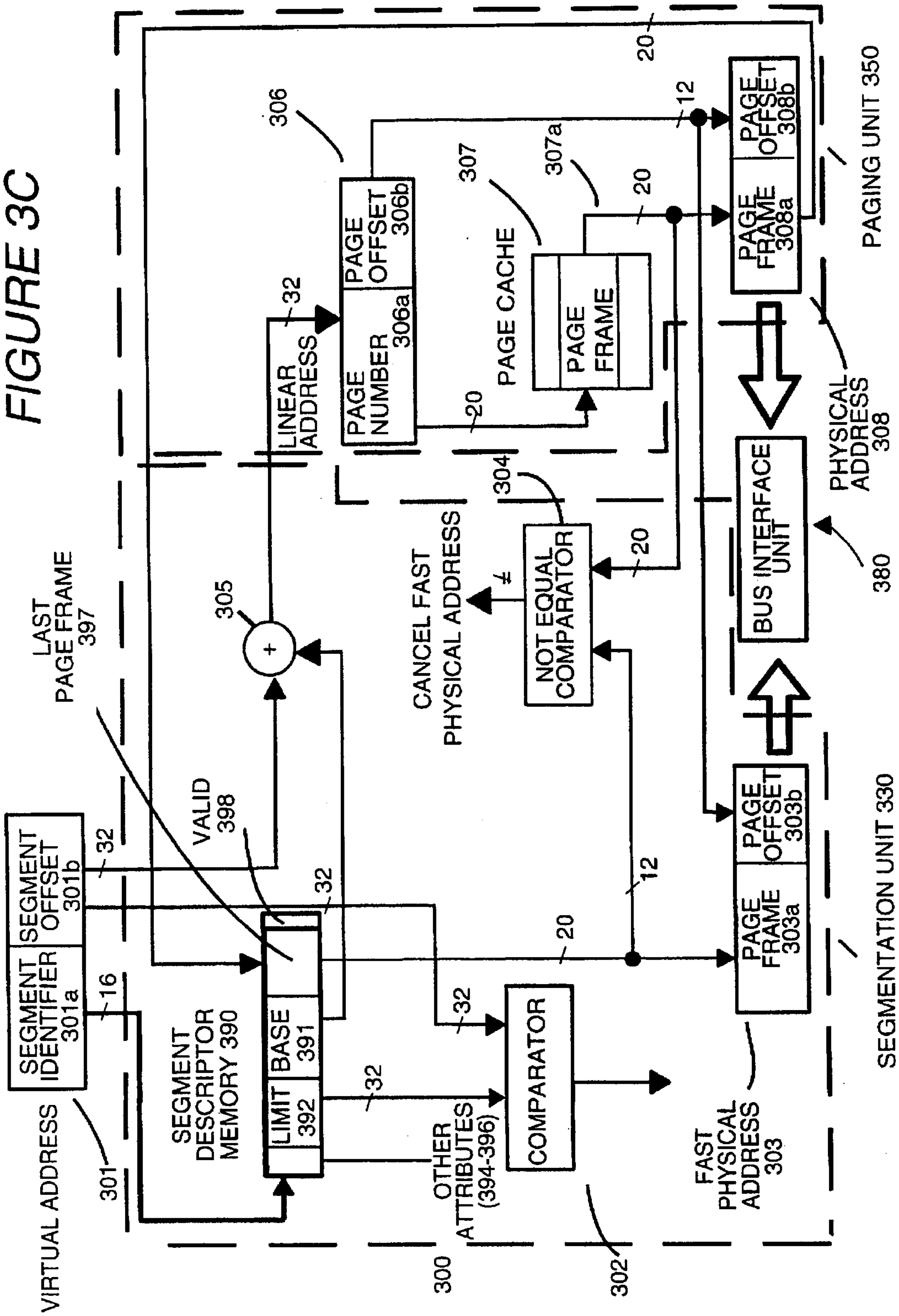

VIRTUAL ADDRESS
301
SEGMENT IDENTIFIER 301a
SEGMENT OFFSET 301b
16
32
LAST PAGE FRAME 397
VALID 398
SEGMENT DESCRIPTOR MEMORY 390
BASE 391
LIMIT 392
OTHER ATTRIBUTES (394-396)
300
305
+
LINEAR ADDRESS
PAGE NUMBER 306a
PAGE OFFSET 306b
306
PAGE CACHE
PAGE FRAME
307
307a
COMPARATOR
302
CANCEL FAST PHYSICAL ADDRESS
310
≠
NOT EQUAL COMPARATOR
304
low 12 bits
12-bit +
overflow
309
FAST PHYSICAL ADDRESS 303
PAGE FRAME 303a
PAGE OFFSET 303b
SEGMENTATION UNIT 330
BUS INTERFACE UNIT
380
PAGE FRAME 308a
PAGE OFFSET 308b
PHYSICAL ADDRESS 308
PAGING UNIT 350
20
12
32

FIGURE 3B

FIGURE 3C


300
VIRTUAL ADDRESS
301
SEGMENT IDENTIFIER 301a
SEGMENT OFFSET 301b
16
32
LAST PAGE FRAME 397
VALID 398
SEGMENT DESCRIPTOR MEMORY 390
LIMIT 392
BASE 391
OTHER ATTRIBUTES (394-396)
COMPARATOR
302
305
+
LINEAR ADDRESS
306
PAGE NUMBER 306a
PAGE OFFSET 306b
307
307a
PAGE CACHE
PAGE FRAME
304
NOT EQUAL COMPARATOR
≠
CANCEL FAST PHYSICAL ADDRESS
20
12
PAGE FRAME 308a
PAGE OFFSET 308b
PAGING UNIT 350
PHYSICAL ADDRESS 308
BUS INTERFACE UNIT
380
PAGE FRAME 303a
PAGE OFFSET 303b
FAST PHYSICAL ADDRESS 303
SEGMENTATION UNIT 330

ADDRESS TRANSLATION METHOD AND MECHANISM USING PHYSICAL ADDRESS INFORMATION INCLUDING DURING A SEGMENTATION PROCESS

FIELD OF THE INVENTION

The invention relates to the field of address translation for memory management in a computer system.

BACKGROUND OF THE INVENTION

Advanced computer hardware systems operate with complex computer software programs. Computer system designers typically separate the virtual address space, the address space used by programmers in their development of software, and the physical address space, the address space used by the computer system. This separation allows programmers to think in terms of their conceptual models, and to design computer software programs without reference to specific hardware implementations. During the actual execution of programs by the computer system, however, these separate addresses must be reconciled by translating software program virtual addresses into actual physical addresses that can be accessed in a computer memory subsystem.

There are many well known approaches for address translation in the memory management mechanism of a computer system. These approaches fall into basically two major categories: those which map the smaller virtual (sometimes called logical, symbolic or user) addresses onto larger physical or real memory addresses, and those which map larger virtual addresses onto smaller physical memory. Translation mechanisms of the former category are employed typically in minicomputers in which relatively small address fields (e.g.: 16 bit addresses) are mapped onto larger real memory. Translation mechanisms of the second category are used typically in microprocessors, workstations and mainframes. Within each of these categories, segmentation only, paging only, and a combination of segmentation and paging are well known for accomplishing the translation process.

The present invention is primarily directed to address translation mechanisms where larger virtual addresses are mapped onto smaller physical addresses, and further to systems where segmentation and optional paging is employed.

In a segmentation portion of an address translation system, the address space of a user program (or programs cooperatively operating as processes or tasks), is regarded as a collection of segments which have common high-level properties, such as code, data, stack, etc. The segmented address space is referenced by a 2-tuple, known as a virtual address, consisting of the following fields: <<s>, <d>>, where <s> refers to a segment number (also called identifier or locator), and <d> refers to a displacement or offset, such as a byte displacement or offset, within the segment identified by the segment number. The virtual address <17,421 >, for example, refers to the 421st byte in segment 17. The segmentation portion of the address translation mechanism, using information created by the operating system of the computer system, translates the virtual address into a linear address in a linear address space.

In a paging portion of an address translation system, a linear (or intermediate) address space consists of a group of pages. Each page is the same size (i.e. it contains the same number of addresses in the linear space). The linear address space is mapped onto a multiple of these pages, commonly, by considering the linear address space as the 2-tuple consisting of the following fields: <<page number >, <page offset >>. The page number (or page frame number) determines which linear page is referenced. The page offset is the offset or displacement, typically a byte offset, within the selected page.

In a paged system, the real (physical) memory of a computer is conceptually divided into a number of page frames, each page frame capable of holding a single page. Individual pages in the real memory are then located by the address translation mechanism by using one or more page tables created for, and maintained by, the operating system. These page tables are a mapping from a page number to a page frame. A specific page may or may not be present in the real memory at any point in time.

Address translation mechanisms which employ both segmentation and paging are well known in the art. There are two common subcategories within this area of virtual address translation schemes: address translation in which paging is an integral part of the segmentation mechanism; and, address translation in which paging is independent from segmentation.

In prior art address translation mechanisms where paging is an integral part of the segmentation mechanism, the page translation can proceed in parallel with the segment translation since segments must start at page boundaries and are fixed at an integer number of pages. The segment number typically identifies a specific page table and the segment offset identifies a page number (through the page table) and an offset within that page. While this mechanism has the advantage of speed (since the steps can proceed in parallel) it is not flexible (each segment must start at a fixed page boundary) and is not optimal from a space perspective (e.g. an integer number of pages must be used, even when the segment may only spill over to a fraction of another page).

In prior art address translation mechanisms where paging is independent from segmentation, page translation generally cannot proceed until an intermediate, or linear, address is first calculated by the segmentation mechanism. The resultant linear address is then mapped onto a specific page number and an offset within the page by the paging mechanism. The page number identifies a page frame through a page table, and the offset identifies the offset within that page. In such mechanisms, multiple segments can be allocated into a single page, a single segment can comprise multiple pages, or a combination of the above, since segments are allowed to start on any byte boundary, and have any byte length. Thus, in these systems, while there is flexibility in terms of the segment/page relationship, this flexibility comes at a cost of decreased address translation speed.

Certain prior art mechanisms where segmentation is independent from paging allow for optional paging. The segmentation step is always applied, but the paging step is either performed or not performed as selected by the operating system. These mechanisms typically allow for backward compatibility with systems in which segmentation was present, but paging was not included.

Typical of the prior art known to the Applicant in which paging is integral to segmentation is the Multics virtual memory, developed by Honeywell and described by the book, "The Multics System", by Elliott Organick. Typical of the prior art known to the Applicant in which optional paging is independent from segmentation is that described in U.S. Pat. No. 5,321,836 assigned to the Intel Corporation, and that described in the Honeywell DPS-8 Assembly

Instructions Manual. Furthermore, U.S. Pat. No. 4,084,225 assigned to the Sperry Rand Corporation contains a detailed discussion of general segmentation and paging techniques, and presents a detailed overview of the problems of virtual address translation.

Accordingly, a key limitation of the above prior art methods and implementations where segmentation is independent from paging is that the linear address must be fully calculated by the segmentation mechanism each time before the page translation can take place for each new virtual address. Only subsequent to the linear address calculation, can page translation take place. In high performance computer systems computer systems, this typically takes two full or more machine cycles and is performed on each memory reference. This additional overhead often can reduce the overall performance of the system significantly.

SUMMARY OF THE INVENTION

An object of the present invention, therefore, is to provide the speed performance advantages of integral segmentation and paging and, at the same time, provide the space compaction and compatibility advantages of separate segmentation and paging.

A further object of the present invention is to provide a virtual address translation mechanism which architecturally provides for accelerating references to main memory in a computer system which employs segmentation, or which employs both segmentation and optional paging.

Another object of the present invention is to provide additional caching of page information in a virtual address translation scheme.

An further object of the present invention is to provide a virtual address translation mechanism which reduces the number of references required to ensure memory access.

According to the present invention, a segmentation unit converts a virtual address consisting of a segment identifier and a segment offset into a linear address. The segmentation unit includes a segment descriptor memory, which is selectable by the segment identifier. The entry pointed to by the segment identifier contains linear address information relating to the specific segment (i.e., linear address information describing the base of the segment referred to by the segment identifier, linear address information describing the limit of the segment referred to by the segment identifier, etc.) as well as physical address information pertaining to the segment—such as the page base of at least one of the pages represented by said segment.

In the above embodiment, unlike prior art systems, both segmentation and paging information are kept in the segmentation unit portion of the address translation system. The caching of this page information in the segmentation unit permits the address translation process to occur at much higher speed than in prior art systems, since the physical address information can be generated without having to perform a linear to physical address mapping in a separate paging unit.

The page base information stored in the segmentation unit is derived from the page frame known from the immediately prior in time address translation on a segment-by-segment basis. In order to complete the full physical address translation (i.e., a page frame number and page offset), the segmentation unit combines the page frame from the segment descriptor memory with the page offset field, and may store this result in a segmentation unit memory, which can be a memory table, or a register, or alternatively, it may generate the full physical address on demand.

This fast physical address generated by the segmentation unit based on the virtual address and prior page information can be used by a bus interface to access a physical location in the computer memory subsystem, even before the paging unit has completed its translation of the linear address into a page frame and page offset. Thus, fewer steps and references are required to create a memory access. Consequently, the address translation step occurs significantly faster. Since address translation occurs in a predominant number of instructions, overall system performance is improved.

The memory access is permitted to proceed to completion unless a comparison of the physical address information generated by the paging unit with the fast physical address generated by the segmentation unit shows that the page frame information of the segmentation unit is incorrect.

In alternative embodiments, the segmentation unit either generates the page offset by itself (by adding the lower portion of the segment offset and the segment base address) or receives it directly from the paging unit.

In further alternate embodiments, the incoming segment offset portion of the virtual address may be presented to the segmentation unit as components. The segmentation unit then combines these components in a typical base-plus-offset step using a conventional multiple input (typically 3-input) adder well known in the prior art.

As shown herein in the described invention, the segment descriptor memory may be a single register, a plurality of registers, a cache, or a combination of cache and register configurations.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a typical prior art virtual address translation mechanism using segmentation and independent paging.

FIG. 2A is a detailed diagram of a typical segment descriptor register of the prior art.

FIG. 2B is a detailed diagram of an embodiment of the present invention, including a portion of a segment descriptor memory used for storing physical address information;

FIG. 3A is a block diagram of an embodiment of the present invention employing segmentation and optional paging, and showing the overall structure and data paths used when paging is disabled during an address translation;

FIG. 3B is a block diagram of the embodiment of FIG. 3A showing the overall structure and data paths used when paging is enabled during an address translation;

FIG. 3C is a block diagram of another embodiment of the present invention, showing an alternative circuit for generating the fast physical address information.

DETAILED DESCRIPTION OF THE INVENTION

General Discussion of Paging & Segmentation

The present invention provides for improved virtual address translation in a computer system. The preferred embodiment employs the invention in a single-chip microprocessor system, however, it is well understood that such a virtual address translation system could be implemented in multiple die and chip configurations without departing from the spirit or claims of the present invention.

Before embarking on a specific discussion of the present invention, however, a brief explanation of the general principles of segmentation and paging follows in order to provide additional background information, and so that the teachings of the present invention may be understood in a proper context.

Referring to FIG. 1, a typical prior art virtual address translation mechanism **100** using both segmentation and, optionally, paging in a computer system is shown. As described in this figure, a data path within the microprocessor transmits a virtual address **101**, consisting of segment identifier **101a** and a segment offset **101b**, to segmentation unit **130**. Segments are defined by segment descriptor entries in at least one segment descriptor table or segment descriptor segment (not shown). Segment descriptor tables are created and managed by the operating system of the computer system, and are usually located in the memory subsystem. Segment descriptor entries are utilized in the CPU of the computer system by loading them into segment descriptor register **190** or a segment descriptor cache (not shown); the segment descriptor register/cache is usually internal to the CPU, and thus more quickly accessible by the translation unit.

In the paging unit **150**, pages are defined by a page table or multiple page tables (not shown), also created and managed by the operating system; again, these tables are also typically located in a memory subsystem. All or a portion of each page table can be loaded into a page cache **107** (within the CPU, sometimes called a translation look-aside buffer) to accelerate page references.

In operation, the segmentation unit **130** first translates a virtual address to a linear address and then (except in the case when optional paging is disabled) paging unit **150** translates the linear address into a real (or physical) memory address.

Typically (as in an x86 microprocessor) the segmentation unit translates a 48-bit virtual address **101** consisting of a 16-bit segment identifier (<s>) **101a** and a 32-bit displacement within that segment (<d>) **101b** to a 32-bit linear (intermediate) address **106**. The 16-bit segment identifier **101a** uniquely identifies a specific segment; this identifier is used to access an entry in a segment descriptor table (not shown). In the prior art, this segment descriptor entry contains a base address of the segment **191**, the limit of the segment **192**, and other attribute information described further below. The segment descriptor entry is usually loaded into a segment descriptor register **190**.

Using adder **105**, the segmentation unit adds the segment base **191** of the segment to the 32-bit segment offset **101b** in the virtual address to obtain a 32-bit linear address. The 32-bit segment offset **101b** in the virtual address is also compared against the segment limit **192**, and the type of the access is checked against the segment attributes. A fault is generated and the addressing process is aborted if the 32-bit segment offset is outside the segment limit, or if the type of the access is not allowed by the segment attributes.

The resulting linear address **106** can be treated as an offset within a linear address space; and in the commonly implemented schemes of the prior art, these offsets are frequently byte offsets. When optional paging is disabled, the linear address **106** is exactly the real or physical memory address **108**. When optional paging is enabled, the linear address is treated as a 2- or 3-tuple depending on whether the paging unit **150** utilizes one or two level page tables.

In the 2-tuple case shown in FIG. 1, which represents single level paging, the linear address, <<p>, <pd>> is divided into a page number field <p > **106a**, and a page displacement (page offset) field within that page (<pd>) **106b**. In the 3-tuple case (not shown) <<dp>, <p>, <pd>>, the linear address is divided into a page directory field (<dp>), a page number field <p> and a page displacement field <pd >. The page directory field indexes a page directory to locate a page table (not shown). The page number field indexes a page table to locate the page frame in real memory corresponding to the page number, and the page displacement field locates a byte within the selected page frame. Thus, paging unit **150** translates the 32-bit linear address **106** from the segmentation unit **130** to a 32-bit real (physical) address **108** using one or two level page tables using techniques which are well known in the art.

In all of the above prior art embodiments where segmentation is independent from paging, the segment descriptor table or tables of the virtual address translator are physically and logically separate from the page tables used to perform the described page translation. There is no paging information in the segment descriptor tables and, conversely, there is no segmentation information in the page tables.

This can be seen in FIG. 2A. In this figure, a typical prior art segment descriptor entry **200**, is shown as it is typically used in a segment descriptor table or segment descriptor register associated with a segmentation unit. As can be seen there, segment descriptor **200** includes information on the segment base **201**, the segment limit **202**, whether the segment is present (P) **203** in memory, the descriptor privilege level (DPL) **204**, whether the segment belongs to a user or to the system (S) **205**, and the segment type **206** (code, data, stack, etc.)

For additional discussions pertaining to the prior art in segmentation, paging, segment descriptor tables, and page tables, the reader is directed to the references U.S. Pat. Nos. 5,408,626, 5,321,836, 4,084,225, which are expressly incorporated by reference herein.

Improved Segmentation Unit Using Paging Information

As shown in the immediate prior art, the paging and segmentation units (circuits) are completely separate and independent. Since the two units perform their translation sequentially, that is, the segment translation must precede the page translation to generate the linear address, high performance computer systems, such as those employing superscalar and superpipelined techniques, can suffer performance penalties. In some cases, it is even likely that the virtual address translation could fall into the systems' "critical path". The "critical path" is a well-known characteristic of a computer system and is typically considered as the longest (in terms of gate delay) path required to complete a primitive operation of the system.

Accordingly, if the virtual address translation is in the critical path, the delays associated with this translation could be significant in overall system performance. With the recognition of this consideration, the present invention includes page information in the segment translation process. The present invention recognizes the potential performance penalty of the prior art and alleviates it by storing paging information in the segmentation unit obtained from a paging unit in previous linear-to-real address translations.

As can be seen in FIG. 2B, the present invention extends the segment descriptor entries of the prior art with a segment entry **290** having two additional fields: a LAST PAGE FRAME field **297** and a VALID field **298**. The LAST PAGE FRAME field **297** is used to hold the high-order 20 bits (i.e.: the page frame) of the real (physical) memory address of the last physical address generated using the specified segment identifier. The VALID field **298** is a 1-bit field, and indicates whether or not the LAST PAGE FRAME field **297** is valid. The remaining fields **291–296** perform the same function as comparable fields **201–206** respectively described above in connection with FIG. 2A.

Segment descriptor tables (not shown) can be located in a memory subsystem, using any of the techniques well-known in the art. As is also known in the art, it is possible

to speed up address translation within the segmentation unit by using a small cache, such as one or more registers, or associative memory. The present invention makes use of such a cache to store segment entries 290 shown above. Unlike the prior art, however, the segment entries 290 of the present invention each contain information describing recent physical address information for the specified segment. Accordingly, this information can be used by a circuit portion of the segmentation unit to generate a new physical address without going through the linear to physical mapping process typically associated with a paging unit.

While in some instances the physical address information may change between two time-sequential virtual addresses to the same segment (and thus, a complete translation is required by both the segmentation and paging units), in the majority of cases the page frame information will remain the same. Thus, the present invention affords a significant speed advantage over the prior art, because in the majority of cases a complete virtual-linear-physical address translation is not required before a memory access is generated.

Embodiment Witb Segmentation & Optional Paging/Paging Disabled

Referring to FIG. 3A, the advantage of using this new information in segment entry 290 in a segmentation unit or segmentation circuit is apparent from a review of the operation of an address translation. In this figure, a paging unit (or paging circuit) is disabled, as for example might occur only when a processor is used in a real mode of operation, rather than a protected mode of operation.

In a preferred embodiment, the present invention employs a segment descriptor memory comprising at least one, and preferably many, segment descriptor registers 390, which are identical in every respect to the segment descriptor register described above in connection with FIG. 2B. These segment descriptor registers are loaded from conventional segment descriptor tables or segment descriptor segments which are well known in the art. Each segment descriptor register 390 is loaded by the CPU before it can be used to reference physical memory. Segment descriptor register 390 can be loaded by the operating system or can be loaded by application programs. Certain instructions of the CPU are dedicated to loading segment descriptor registers, for example, the "LDS" instruction, "Load Pointer to DS Register". Loading by the operating system, or execution of instructions of this type, causes a base 391, limit 392, descriptor privilege level 394, system/user indicator 395, and type 396 to be loaded from segment tables or segment descriptor segments as in the prior art. The three remaining fields are present 393, LAST PAGE FRAME 397 and VALID 398. When a segment descriptor register 390 is loaded, present 393 is set to 1, indicating that the segment descriptor register 390 contents are present; the valid field 398 is set to 0, indicating that the last page frame number field 397 is not valid; and the LAST PAGE FRAME field 397 is not set, or may be set to 0.

After the loading of a segment descriptor register 390, instructions of the CPU may make references to virtual memory; if a segment descriptor register is referenced before it is loaded, as indicated by present field 393 set to 0, a fault occurs and the reference to the segment descriptor register is aborted.

As explained above, the CPU makes references to virtual memory by specifying a 48-bit virtual address, consisting of a 16-bit segment identifier 301*a* and a 32-bit segment offset 301*b*. A data path within the CPU transmits virtual address 301 to the address translation mechanism 300.

Segment descriptor memory 390 is indexed by segment number, so each entry in this memory containing data characteristics (i.e., base, access rights, limit) of a specific segment is selectable by the segment identifier from the virtual address. Assuming this is the first reference to physical memory specifying a newly loaded segment descriptor register, since the VALID bit 398 is set to false, a prior art virtual address translation takes place. This involves, among other things, as explained earlier, various validity checks (including checking attributes 394–396, segment limit checking using comparator 302 and potentially others), and using adder 305 to add the segment descriptor's base address 391 to the segment offset 301*b* to calculate a linear address 306.

While the implementation in the embodiment of FIG. 3A shows the addition of the base address 391 to the segment offset 301*b* using adder 305 to generate the linear address 306, it will be understood by those skilled in the art that this specific implementation of the virtual to linear address translation is not the only implementation of the present invention. In other implementations, the segment offset 301*b* might consist of one or more separate components. Different combinations of one or more of these components might be combined using well known techniques to form a linear address, such as one utilizing a three-input adder. The use of these components is discussed, for example, in U.S. Pat. No. 5,408,626, and that description is incorporated by reference herein.

As is well known, in this embodiment where paging is disabled, linear address 306 is also a physical address which can be used as the physical address 308. Memory access control operations are not shown explicitly since they are only ancillary to the present invention, and are well described in the prior art. In general, however, a bus interface unit 380 is typically responsible for interactions with the real (physical) memory subsystem. The memory subsystem of a computer system employing the present invention preferably has timing and address and data bus transaction details which are desirably isolated from the CPU and address translation mechanism. The bus interface unit 380 is responsible for this isolation, and can be one of many conventional bus interface units of the prior art.

In the present invention, bus interface unit 380 receives the real memory address 308 from address translation mechanism 300 and coordinates with the real memory subsystem to provide data, in the case of a memory read request, or to store data, in the case of a memory write request. The real memory subsystem may comprise a hierarchy of real memory devices, such as a combination of data caches and dynamic RAM, and may have timing dependencies and characteristics which are isolated from the CPU and address translation mechanism 300 of the computer system by the bus interface unit 380.

Simultaneous with the first memory reference using the calculated physical address 308, the LAST PAGE FRAME field of the selected segment descriptor register 390 is loaded with the high-order 20 bits of the physical address, i.e.: the physical page frame, and the VALID bit is set to indicate a valid state. This paging information will now be used in a next virtual address translation.

Accordingly, when a next, new virtual address 301 is to be translated, the entry selected from segment descriptor memory 390 will likely contain the correct physical frame page number (in the LAST PAGE FRAME field 397). Thus, in most cases, the base physical address in memory for the next, new referenced virtual address will also be known from a previous translation.

The first step of the virtual address translation, therefore, is to determine if a FAST PHYSICAL ADDRESS 303 can

be used to begin a fast physical memory reference. Adder **309**, a 12-bit adder, adds the low-order 12 bits of the segment offset **301***b* of virtual address **301** to the low-order 12-bits of base **391** of the segment entry in segment descriptor register **390** referenced by the segment identifier **301***a*. This addition results in a page offset **303***b*. In parallel with adder **309**, 32-bit adder **305** begins a full 32-bit add of segment base **391** and segment offset **301***b*, to begin producing the linear address; however, this full 32-bit add will obviously require more time. In the preferred embodiment, adder **309** is a separate 12-bit adder; however, it should be noted that adder **309** also could be implemented as the low order 12-bits of 32-bit adder **305**.

Simultaneous with the beginning of these two operations, VALID bit **398** is inspected. If VALID bit **398** is set to 1, as soon as 12-bit adder **309** has completed, 20-bit LAST PAGE FRAME **397** is concatenated with the result of adder **309** to produce FAST PHYSICAL ADDRESS **303**, consisting of a page frame number **303***a*, and page offset **303***b*. FAST PHYSICAL ADDRESS **303** then can be used to tentatively begin a reference to the physical memory. It should be understood that the FAST PHYSICAL ADDRESS **303** transmitted to bus interface unit **380** could also be stored in a register or other suitable memory storage within the CPU.

In parallel with the fast memory reference, limit field **392** is compared to the segment offset **301***b* of the virtual address by comparator **302**. If the offset in the virtual address is greater than the limit, a limit fault is generated, and virtual address translation is aborted.

Also in parallel with the fast memory reference, adder **305** completes the addition of base **391** to the segment offset field **301***b* of virtual address to produce linear address (in this case physical address also) **306**. When this calculation is completed, the page frame number **308***a* of physical address **308** is compared to LAST PAGE FRAME **397** by Not Equal Comparator **304**. If page frame **308***a* is unequal to the LAST PAGE FRAME **397**, or if 12-bit Adder **309** overflowed (as indicated by a logic "1" at OR gate **310**), the fast memory reference is canceled, and the linear address **306**, which is equal to the physical address **308**, is used to begin a normal memory reference. If page frame **308***a* is equal to LAST PAGE FRAME **397**, and 12-bit Adder **309** did not overflow (the combination indicated by a logic "0" at the output of OR gate **310**), the fast memory reference is allowed to fully proceed to completion.

After any fast memory reference which is cancelled by the CANCEL FAST PHYSICAL ADDRESS signal output of OR gate **310**, page frame **308***a* is loaded into the LAST PAGE FRAME **397** in the segment descriptor memory **390** for subsequent memory references.

Depending on the particular design desired, it should also be noted that writes to the memory, or reads which cause faults using FAST PHYSICAL ADDRESS **303** may be pended since the FAST PHYSICAL ADDRESS **303** may prove to be invalid.

Accordingly, it can be seen that the parallel physical address calculation undertaken by the improved segmentation unit of the present invention generates a faster physical memory access than possible with prior art systems.

Embodiment With Segmentation & Paging/Paging Enabled

The present invention can also be used with address translation units using paging enabled, as can be seen in the embodiments of FIGS. 3B and 3C.

In the embodiment of FIG. 3B, the same segmentation unit structure **300** as that shown in FIG. 3A is used, and the operation of segmentation unit **300** is identical to that already explained above. As before, segment descriptor memory (registers) **390** are loaded from conventional segment descriptor tables or segment descriptor segments, using one or more of the procedures described above. First, the base **391** limit **392** descriptor privilege level **394**, system/user indicator **395**, and type **396** are loaded from segment tables or segment descriptor segments as explained earlier. When segment descriptor register **390** is loaded, present **393** is set to 1, indicating that the segment descriptor register **390** contents are present; the valid field **398** is set to 0, indicating that the last page frame number field **397** is not valid; and the LAST PAGE FRAME field **397** is not set, or may be set to 0.

As explained above, after the loading of a segment descriptor register **390**, instructions of the CPU may make references to virtual memory; if a segment descriptor register is referenced before it is loaded, as indicated by present field **393** set to 0, a fault occurs and the reference to the segment descriptor register is aborted.

As further explained above, the 48 bit virtual address **301** (consisting of a 16 bit segment identifier **301***a* and a 32 bit segment offset **301***b*) is transmitted by a data path to segmentation unit **300**, and an index into segment descriptor memory **390** is performed to locate the specific segment descriptor for the segment pointed to by segment identifier **301***a*. Assuming this is the first reference to physical memory specifying a newly loaded segment descriptor register, since the VALID bit is set to false, a prior art virtual address translation takes place. This involves, among other things, as explained earlier, various validity checks (including checking attributes **394–396**, segment limit checking using comparator **302** and potentially others), and using adder **305** to add the segment descriptor's base address **391** to the segment offset **301***b* to calculate a linear address **306**.

As is well known, in this configuration where paging is enabled, linear address **306** must undergo a further translation by paging unit **350** to obtain the physical address **308** in the memory subsystem. In the preferred embodiment of the invention, looking first at FIG. 3B, the output of adder **305** will be a 32-bit linear address, corresponding to a 20-bit page number **306***a* and a 12-bit page offset **306***b*. Typically, the page number **306***a* is then indexed into a page descriptor table (not shown) to locate the appropriate page frame base physical address in memory. These page descriptor tables are set up by the operating system of the CPU using methods and structures well known in the art, and they contain, among other things, the base physical address of each page frame, access attributes, etc.

However, in most systems, including the present invention, a page cache **307** is used in order to hold the physical base addresses of the most recently used page frames. This cache can take the form of a table, associative cache, or other suitable high speed structure well known in the art. Thus, page number **306***a* is used to access page data (including physical base addresses for page frames) in an entry in page cache **307**.

If page cache **307** hits, two things happen: first, a 20-bit PAGE FRAME **307***a* (the page frame in physical memory) replaces the high-order 20 bits (page number **306***a*) of the linear address **306**, and, when concatenated with the page offset **306***b* results in a real (physical) address **308**, which is used to perform a memory access through bus interface unit **380** along the lines explained above. Second, newly generated page frame **308***a* is also stored in segment descriptor memory **390** in the selected LAST PAGE FRAME field **397** to be used for a fast access in the next address translation. When LAST PAGE FRAME field **397** is stored, selected

VALID bit 398 is set to 1 to indicate that LAST PAGE FRAME 397 is valid for use.

In the event of a page cache miss, the appropriate page frame number 308a is located (using standard, well-known techniques) to generate physical address 308, and is also loaded into segment descriptor memory 390 in the selected LAST PAGE FRAME field 397. The selected VALID bit 398 is also set to indicate a valid state. Thus, there is paging information in the segmentation unit that will now be used in the next virtual address translation.

When a next, new virtual address 301 is to be translated, the segment identifier 301a will likely be the same as that of a previously translated virtual address, and the entry selected from segment descriptor memory 390 will also likely contain the correct physical frame (in LAST PAGE FRAME field 397) from the previous translation. As with the above embodiment, one or more registers, or a cache may be used for the segment descriptor memory 390.

The first step then determines if a FAST PHYSICAL ADDRESS 303 can be used to begin a fast physical memory reference. Adder 309, a 12-bit adder, adds the low-order 12 bits of the segment offset 301b of virtual address 301 to the low-order 12-bits of base 391 of the segment entry in segment descriptor register 390 referenced by the segment identifier 301a. This addition results in a page offset 303b. In parallel with adder 309, 32-bit adder 305 begins a full 32-bit add of segment base 301 and segment offset 301b, to begin producing the linear address; however, this full 32-bit add will obviously require more time. In the preferred embodiment, adder 309 is a separate 12-bit adder; however, it should be noted that adder 309 also could be implemented as the low order 12-bits of 32-bit adder 305.

Simultaneous with these beginning of these two operations, VALID bit 398 is inspected. If VALID bit 398 is set to 1, as soon as 12-bit adder 309 has completed, 20-bit LAST PAGE FRAME 397 is concatenated with the result of adder 309 to produce FAST PHYSICAL ADDRESS 303, consisting of a page frame number 303a, and page offset 303b. FAST PHYSICAL ADDRESS 303 then can be used to tentatively begin a reference to the physical memory. Again, it should be understood that the FAST PHYSICAL ADDRESS 303 transmitted to bus interface unit 380 could also be stored in a register or other suitable memory storage within the CPU.

As before, limit field 302 is compared to the segment offset 301b of the virtual address by comparator 302. If the offset in the virtual address is greater than the limit, a limit fault is generated, and virtual address translation is aborted.

This new virtual address is also translated by paging unit 350 in the same manner as was done for the previous virtual address. If page cache 307 hits based on the page number 306a, two things happen: first, a 20-bit PAGE FRAME 307a (the page frame in physical memory) replaces the high-order 20 bits (age number 306a) of the linear address 306, and, when concatenated with the page offset 306b results in a physical address 308. This real address may or may not be used, depending on the result of the following: in parallel with the aforementioned concatenation, the PAGE FRAME 307a, is compared to LAST PAGE FRAME 397 from the segment descriptor memory 390 by Not Equal Comparator 304. The result of Not Equal Comparator (that is, the Not Equal condition) is logically ORed with the overflow of 12-bit adder 309 by OR gate 310. If the output of OR gate 310 is true (i.e. CANCEL FAST PHYSICAL ADDRESS is equal to binary one), or if PAGE CACHE 307 indicates a miss condition, the fast memory reference previously begun is canceled, since the real memory reference started is an invalid reference. Otherwise, the fast memory reference started is allowed to fully proceed to completion, since it is a valid real memory reference.

If CANCEL FAST PHYSICAL ADDRESS is logical true, it can be true for one of two, or both reasons. In the case that Or gate 310 is true, but page cache 307 indicates a hit condition, physical address 308 is instead used to start a normal memory reference. This situation is indicative of a situation where LAST PAGE FRAME 397 is different from the page frame 308a of the current reference.

In the case that page cache 307 did not indicate a hit, a page table reference through the page descriptor table is required and virtual address translation proceeds as in the prior art. The page frame 308a information is again stored in the LAST PAGE FRAME field 397 in the segment descriptor memory 390 for the next translation.

Also, after any fast memory reference which is canceled by the CANCEL FAST PHYSICAL ADDRESS signal output of OR gate 310, page frame number 308a is loaded into the LAST PAGE FRAME 397 in the segment descriptor memory 390 for subsequent memory references.

Depending on the particular design desired, it should also be noted that in this embodiment also, writes to the memory, or reads which cause faults using FAST PHYSICAL ADDRESS 303 may be pended since the FAST PHYSICAL ADDRESS 303 may prove to be invalid.

The alternative embodiment shown in FIG. 3C is identical in structure and operation to the embodiment of FIG. 3B, with the exception that the 12-bit Adder 309 is not employed. In this embodiment, the segmentation unit 330 does not create the lower portion (age offset 303a) of the fast physical address in this manner. Instead, the page offset 306a resulting from 32-bit adder 305 is used.

It can be seen that the present invention has particular relevance to computers using sequential type of segmentation and paging translation, such as the X86 family of processors produced by the Intel Corporation (including the Intel 80386, Intel 80486 and the Intel Pentium Processor), other X86 processors manufactured by the NexGen Corporation, Advanced Micro Devices, Texas Instruments, International Business Machines, Cyrix Corporation, and certain prior art computers made by Honeywell. These processors are provided by way of example, only, and it will be understood by those skilled in the art that the present invention has special applicability to any computer system where software executing on the processors is characterized by dynamic execution of instructions in programs in such a way that the virtual addresses are generally logically and physically located near previous virtual addresses.

The present invention recognizes this characteristic, employing acceleration techniques for translating virtual to real addresses. In particular, the present invention utilizes any of the commonly known storage structures (specific examples include high speed registers and/or caches) to store previous address translation information, and to make this previous address translation information available to the system whenever the next subsequent reference relies on the same information. In this way, the system can utilize the previously stored information from the high speed storage to begin real memory references, rather than be forced to execute a more time consuming translation of this same information, as was typically done in the prior art.

As will be apparent to those skilled in the art, there are other specific circuits and structures beyond and/or in addition to those explicitly described herein which will serve to implement the translation mechanism of the present invention. Finally, although the above description enables the

specific embodiment described herein, these specifics are not intended to restrict the invention, which should only be limited as defined by the following claims.

I claim:

1. An address translation system for translating a virtual address having a segment identifier and an offset field into a physical address, said address translation system being used in a computer system and comprising:

(a) a segmentation unit for generating linear address information based on the virtual address information, and for storing first physical address information having a first physical page frame and a first physical page offset; and

(b) a paging unit coupled to the segmentation unit for generating said first physical page frame and second physical address information having a second physical page frame and a second page offset; and

(c) said segmentation unit further including:

a segment descriptor memory, selectable by said segment identifier of said virtual address, and capable of storing

I) linear address information describing the base of the segment,

ii) linear address information describing the limit of the segment,

iii) said first physical page frame;

a limit comparator for comparing whether the offset field exceeds the limit of the segment; and

a physical address comparator for comparing the first physical page frame and the second physical page frame

wherein a system memory access can be made by said computer system based on said first physical address information, and wherein the memory access is canceled if the first physical page frame and second physical page frame are not equal.

2. The system of claim 1, wherein the paging unit further includes a page cache for storing page frames of physical pages most recently used by the computer system.

3. The system of claim 1, wherein the segment descriptor memory comprises a register and/or cache, and stores a plurality of first physical page frames associated with a plurality of segments.

4. The system of claim 1, wherein the first physical page frame generated by the paging unit is stored in the segment descriptor memory of the segmentation unit and can be used in a subsequent virtual to physical address translation.

5. The system of claim 1, wherein the segment descriptor memory is further capable of storing:

(iv) information describing whether said page frame can be used for an address translation.

6. The system of claim 1, wherein the segmentation unit also uses either or both index and displacement information to generate the linear address information.

7. An address translation system for translating a virtual address having a segment identifier and an offset field into a physical address, said address translation system being used in a computer system and comprising:

(a) a segmentation unit for generating linear address information based on the virtual address information, and for storing first physical address information having a first physical page frame and a first physical page offset; and

(b) a paging unit coupled to the segmentation unit for generating said first physical page frame and second physical address information having a second physical page frame and a second page offset, and

(c) said segmentation unit further including:

a segment descriptor memory, selectable by said segment identifier of said virtual address, and capable of storing

i) linear address information describing the base of the segment,

ii) linear address information describing the limit of the segment,

iii) said first physical page frame;

a limit comparator for comparing whether the offset field exceeds the limit of the segment; and

an adder, wherein the first physical page offset can be generated by adding a portion of the virtual address offset field and a portion of the segment base in the segment descriptor memory;

wherein a system memory access can be made by said computer system based on said first physical address information.

8. An address translation system for translating a virtual address having a segment identifier and an offset field into a physical address said address translation system being used in a computer system and comprising:

(a) a segmentation unit for generating linear address information based on the virtual address information and for storing first physical address information having a first physical page frame and a first physical page offset, and

(b) a paging unit coupled to the segmentation unit for generating said first physical page frame and second physical address information having a second physical page frame and a second page offset, and

(c) said segmentation unit further including:

a segment descriptor memory, selectable by said segment identifier of said virtual address, and capable of storing

I) linear address information describing the base of the segment,

ii) linear address information describing the limit of the segment,

iii) said first physical page frame;

wherein the first physical page offset can be generated by adding the virtual address offset field and the segment base in the segment descriptor memory; and

a limit comparator for comparing whether the offset field exceeds the limit of the segment;

wherein a system memory access can be made by said computer system based on said first physical address information.

9. A system for address translation in a CPU comprising:

a) an instruction set employing virtual addresses for accessing data or instructions located at physical addresses in a memory subsystem, said virtual addresses containing a segment identifier field selecting a segment and a segment offset field selecting an offset within the selected segment;

b) a segmentation circuit for generating linear addresses based on the virtual addresses, and for storing physical address information, said segmentation circuit including a physical address comparator and at least one segment descriptor memory selectable by said segment identifier, said segment descriptor memory capable of storing:

(i) segment data describing a segment, said segment data including linear address information describing the base and limit of the segment in a linear address space;

(ii) a physical address information corresponding to a page frame; and

c) a paging circuit for receiving the linear addresses and generating physical addresses, said paging circuit including a page cache, said page cache optionally storing page frame information for said CPU; and

d) a bus interface, capable of coupling the segmentation and paging circuits of the CPU to the memory subsystem, and for performing memory accesses in response to physical address information from either of said segmentation and paging circuits; and

wherein the physical address comparator compares the page frame in the segmentation circuit descriptor memory with the page frame in the page cache, and wherein the memory access is canceled if the page frame in the segmentation circuit segment descriptor memory is different from the page frame in the page cache.

10. The system of claim **9**, wherein the page frame generated by the paging unit is stored in the segmentation circuit.

11. The system of claim **9**, wherein the segment descriptor cache is further capable of storing:

(iv) information describing whether said physical address information can be used for an address translation.

12. The system of claim **9**, wherein the segmentation circuit uses index and/or displacement information to generate the linear addresses.

13. A system for address translation in a CPU comprising:

a) an instruction set employing virtual addresses for accessing data or instructions located at physical addresses in a memory subsystem said virtual addresses containing a segment identifier field selecting a segment and a segment offset field selecting an offset within the selected segment;

b) a segmentation circuit for generating linear addresses based on the virtual addresses, and for storing physical address information, said segmentation circuit including an adder and at least one segment descriptor memory selectable by said segment identifier, said segment descriptor memory capable of storing:

(i) segment data describing a segment, said segment data including linear address information describing the base and limit of the segment in a linear address space;

(ii) a physical address information corresponding to a page frame,

c) a paging circuit for receiving the linear addresses and generating physical addresses said paging circuit including a page cache, said page cache optionally storing page frame information for said CPU, and

wherein a page offset is generated by adding a portion of the segment offset field to a portion of the segment base in the segment descriptor memory; and

d) a bus interface, capable of coupling the segmentation and paging circuits of the CPU to the memory subsystem and for performing memory accesses in response to physical address information from either of said segmentation and paging circuits.

14. An address translation system for translating a virtual address having a segment and an offset field into a physical address, said address translation system being used in a computer system and comprising:

a) a segmentation unit for generating linear address information based on the virtual address information, and for generating and storing a physical address including a page frame and a page offset, said segmentation unit further including:

b) a segment descriptor memory, selectable by said segment identifier of said virtual address, and capable of storing

i) linear address information describing the base of the segment.

ii) linear address information describing the limit of the segment,

iii) said page frame,

a physical address memory for storing said physical address including said page frame and a page offset; and

a physical address comparator for comparing the page frame from said virtual address with a page frame corresponding to a different, later in time virtual address;

wherein an access to a memory subsystem can be made by said computer system based on said physical address, and wherein the memory access is canceled if the page frames of said different virtual addresses are not equal.

15. An address translation system for translating a virtual address having a segment and an offset field into a physical address, said address translation system being used in a computer system and comprising:

a) a segmentation unit for generating linear address information based on the virtual address information, and for generating and storing a physical address including a page frame and a page offset, said segmentation unit further including:

b) a segment descriptor memory, selectable by said segment identifier of said virtual address, and capable of storing

i) linear address information describing the base of the segment,

ii) linear address information describing the limit of the segment,

iii) said page frame;

a physical address memory for storing said physical address including said page frame number and a page offset; and

an adder; and

wherein the page offset is generated by adding a portion of the virtual address offset field and a portion of the segment base in the segment descriptor memory, and is then stored in a register corresponding to the physical address memory;

wherein an access to a memory subsystem can be made by said computer system based on said physical address.

16. The system of claim **15**, wherein the segment descriptor memory includes at least a register and/or a cache and stores a plurality of first physical page frames associated with a plurality of segments.

17. The system of claim **15**, further including a bus interface capable of coupling the segmentation unit to the memory subsystem, and for performing memory accesses in response to physical address information from said segmentation unit.

18. The system of claim **17**, wherein said bus interface provides either separate address/data lines to said memory, or multiplexed address/data lines.

19. The system of claim **15**, wherein the segment descriptor memory is further capable of storing:

(iv) information describing whether said page frame can be used for an address translation.

20. The system of claim 15, wherein the segmentation unit also uses either or both index and displacement information to generate the linear address information.

21. A method of calculating physical addresses from virtual addresses, said method comprising:

a) calculating a first physical address, having a first page frame field and a first page offset field based on a virtual address;

b) storing said first page frame field of said first physical address;

c) calculating a second physical address based on a second virtual address including a second page frame field and a second page offset field;

d) generating a third physical address based on the first page frame field and the second page offset field;

e) generating a memory access request based on said third physical address;

(f) canceling said access request to memory using said third physical address if the first page frame field is not equal to the second page frame field of said second physical address.

22. The method of claim 21, wherein the page frame fields most recently used by the computer system are stored.

23. The method of claim 21, further including a step: checking whether the first page frame field can be used for an address translation.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 5,895,503

DATED : April 20, 1999

INVENTOR(S) : Richard A. Belgard

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

On the title page item [76],

the inventor's address information should be:

Richard A. Belgard, 15571 Peach Hill Road, Saratoga, CA 95070

On the front page, the following references should be included under "References Cited" in the US PATENT DOCUMENTS section:

4,400,774	8/23	Toy	364/200
5,165,028	11/92	Zulian	395/400

On the front page, the following reference should be included under "References Cited" in the FOREIGN PATENT DOCUMENTS section:

0668565	8/95	European Patent Office

COLUMN 7, LINE 20	"Witb" should be -- With --
COLUMN 9, LINE 60	"Witb" should be --With --

Signed and Sealed this

Twenty-first Day of December, 1999

Attest:

Q. TODD DICKINSON

Attesting Officer

Acting Commissioner of Patents and Trademarks

EXHIBIT C

US006226733B1

(12) **United States Patent**
Belgard

(10) **Patent No.: US 6,226,733 B1**
(45) **Date of Patent: May 1, 2001**

(54) **ADDRESS TRANSLATION MECHANISM AND METHOD IN A COMPUTER SYSTEM**

(76) Inventor: **Richard A. Belgard**, 21250 Glenmont Dr., Saratoga, CA (US) 95070

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **08/905,356**

(22) Filed: **Aug. 4, 1997**

Related U.S. Application Data

(63) Continuation of application No. 08/458,479, filed on Jun. 2, 1995, now Pat. No. 5,895,503.

(51) **Int. Cl.**[7] **G06F 12/10**
(52) **U.S. Cl.** **711/213**; 711/206; 711/207; 711/208; 711/209; 711/217; 711/218; 711/219; 711/220
(58) **Field of Search** 711/3, 213–220, 711/200–209; 712/233–238

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,084,225		4/1978	Anderson et al.	711/206
4,400,774	*	8/1983	Toy	711/3
5,165,028		11/1992	Zulian	711/3
5,321,836	*	6/1994	Crawford et al.	711/206
5,335,333		8/1994	Hinton et al.	711/207
5,423,014		6/1995	Hinton et al.	711/3

FOREIGN PATENT DOCUMENTS

0668565 8/1995 (EP) .

OTHER PUBLICATIONS

Computer Architecture A Quantitative Approach, Hennessey and Patterson, pp. 432–497.
U5S 486 Green CPU, United Microelectronics Corporation, 1994–95, pp. 3–1 to 3–26.
The Multics System, Elliot Organick, 1972, pp. 6–7, 38–51.
DPS–8 Assembly Instructions, Honeywell Corporation, Apr., 1980, Chapters 3 and 5.

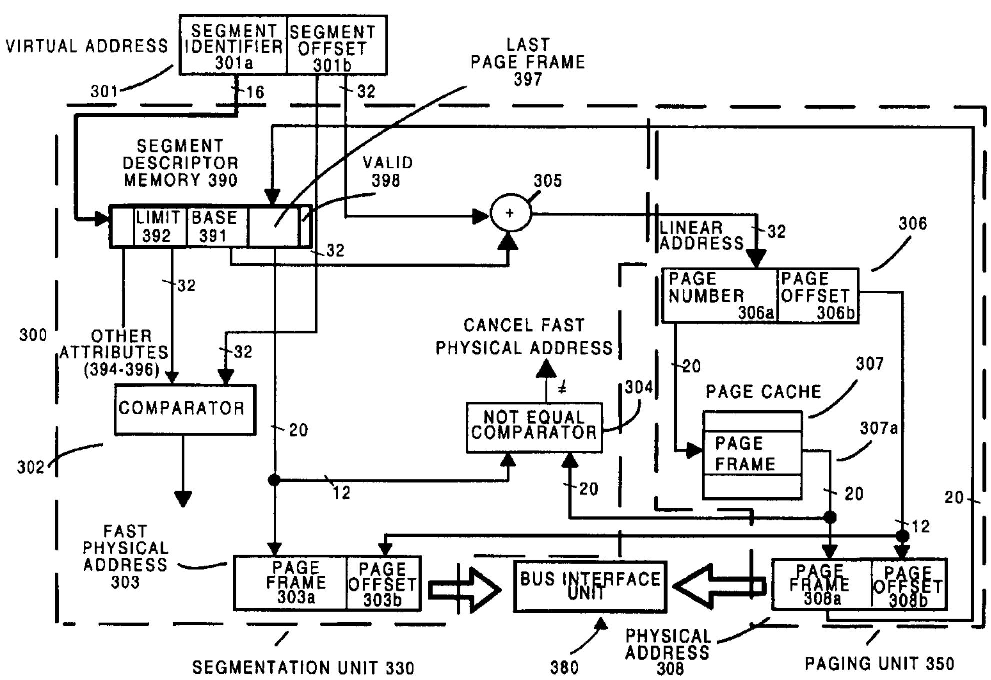
* cited by examiner

Primary Examiner—Than Nguyen
(74) *Attorney, Agent, or Firm*—Law +

(57) **ABSTRACT**

An improved address translation method and mechanism for memory management in a computer system is disclosed. A fast physical address is generated in parallel with a fully computed virtual-linear-physical address in a system using segmentation and optional paging. This fast physical address is used for a tentative or speculative memory reference, which reference can be canceled in the event the fast physical address does not match the fully computed address counterpart. In this manner, memory references can be accelerated in a computer system by avoiding a conventional translation scheme requiring two separate and sequential address translation operations—i.e. from virtual to linear, and from linear to physical.

74 Claims, 5 Drawing Sheets



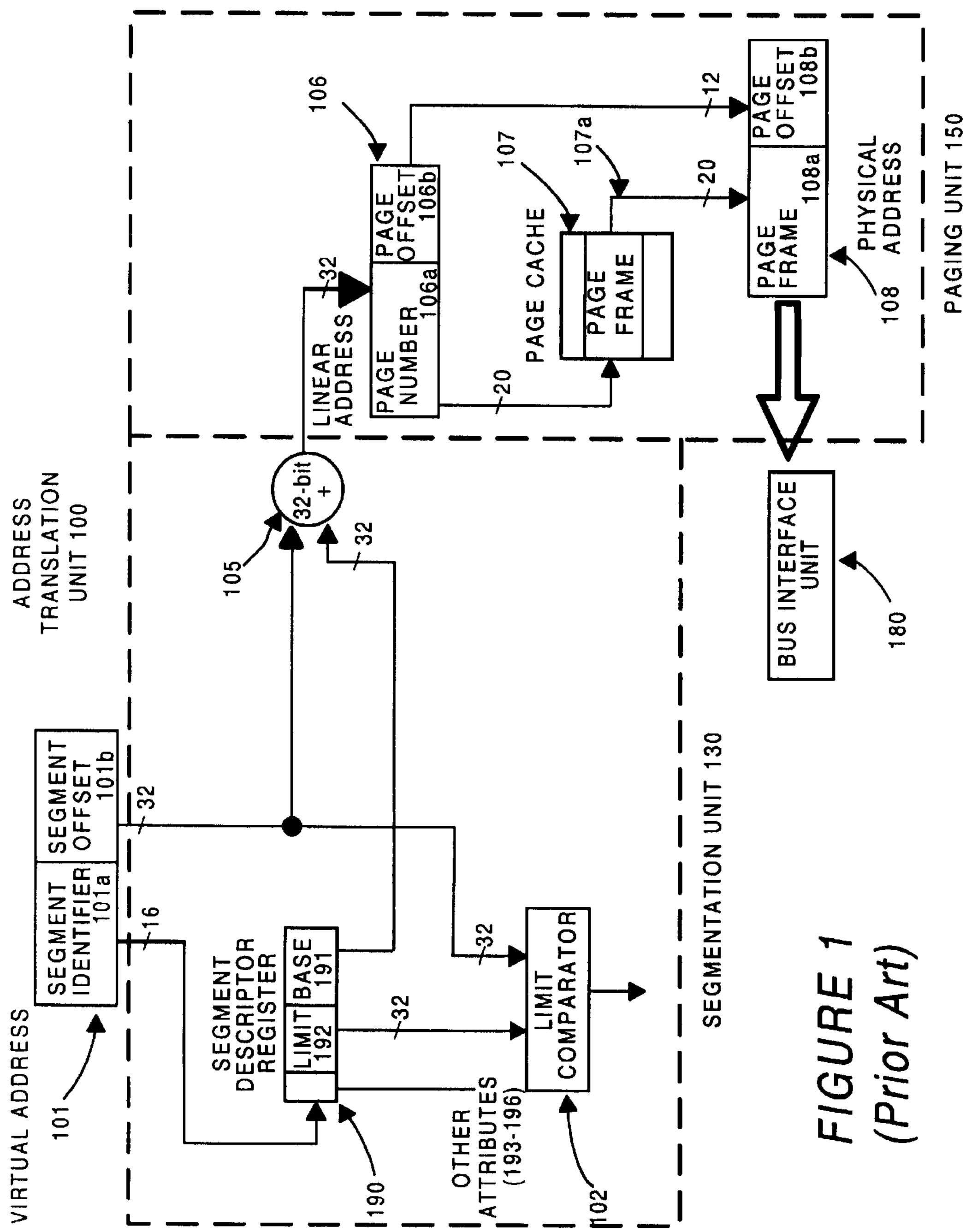
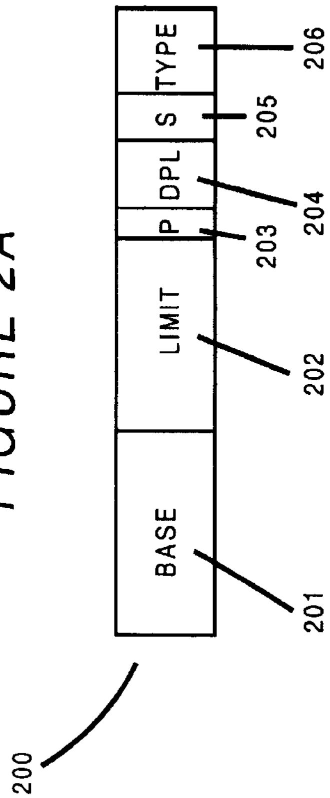


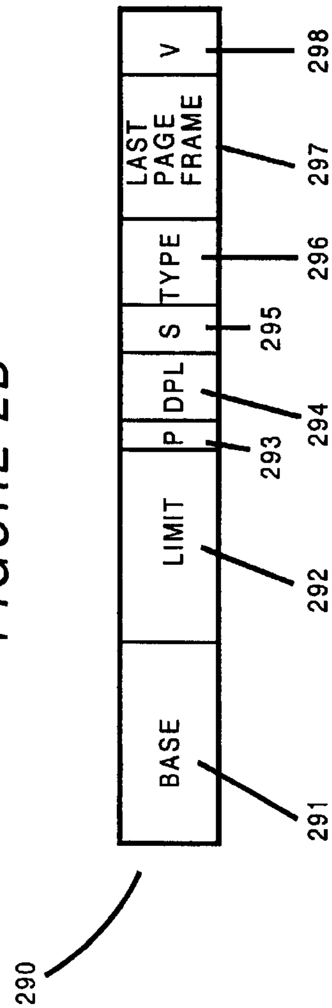
FIGURE 1
(Prior Art)

FIGURE 2A



PRIOR ART SEGMENT DESCRIPTOR REGISTER

FIGURE 2B



SEGMENT DESCRIPTOR MEMORY

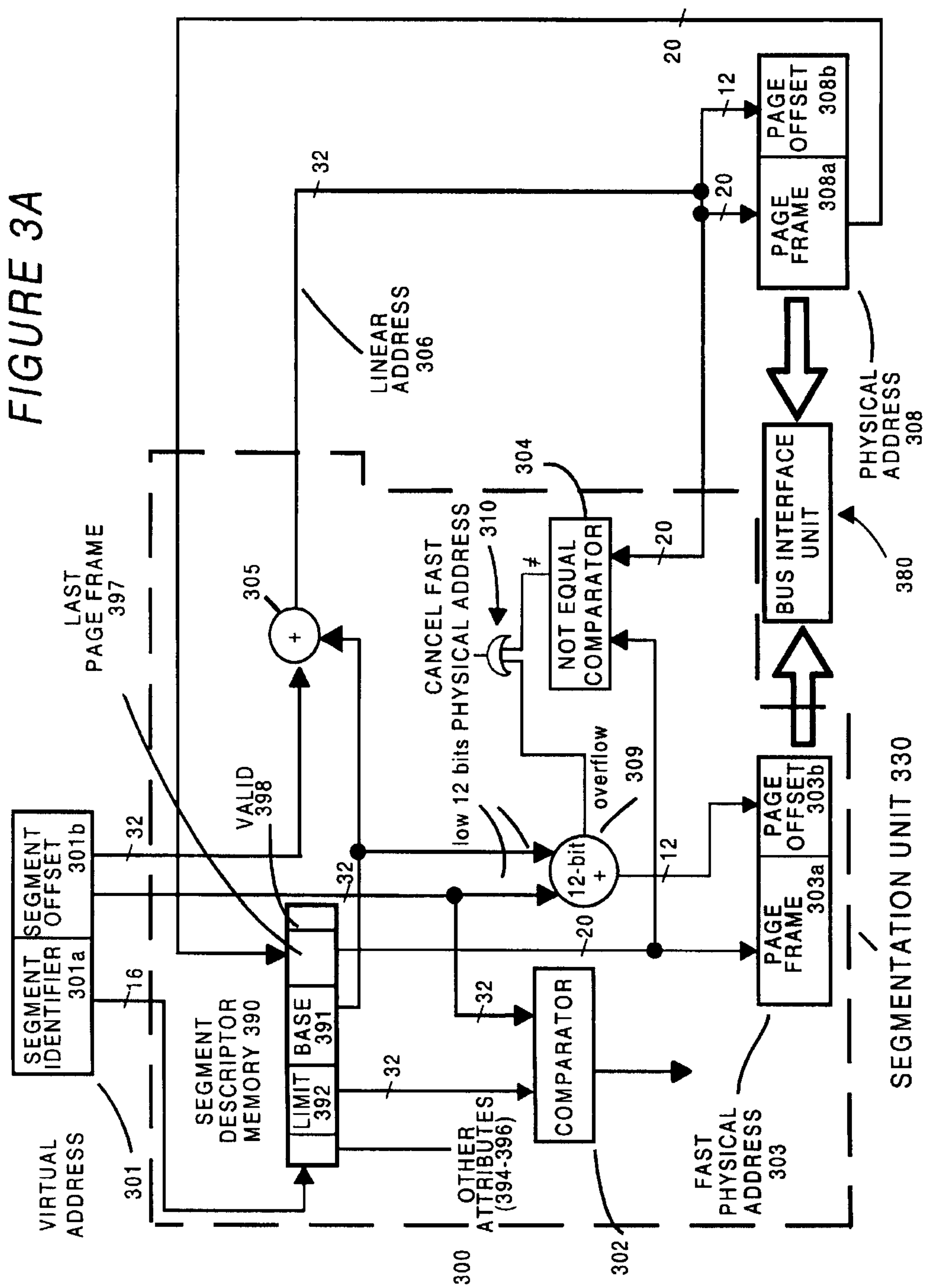

VIRTUAL ADDRESS 301
SEGMENT IDENTIFIER 301a
SEGMENT OFFSET 301b
16
32
SEGMENT DESCRIPTOR MEMORY 390
LIMIT 392
BASE 391
VALID 398
LAST PAGE FRAME 397
OTHER ATTRIBUTES (394-396)
COMPARATOR
302
300
305
+
LINEAR ADDRESS 306
CANCEL FAST PHYSICAL ADDRESS 310
low 12 bits PHYSICAL ADDRESS
12-bit +
overflow 309
≠
NOT EQUAL COMPARATOR
304
20
12
PAGE FRAME 303a
PAGE OFFSET 303b
FAST PHYSICAL ADDRESS 303
SEGMENTATION UNIT 330
BUS INTERFACE UNIT
380
PAGE FRAME 308a
PAGE OFFSET 308b
PHYSICAL ADDRESS 308

FIGURE 3A

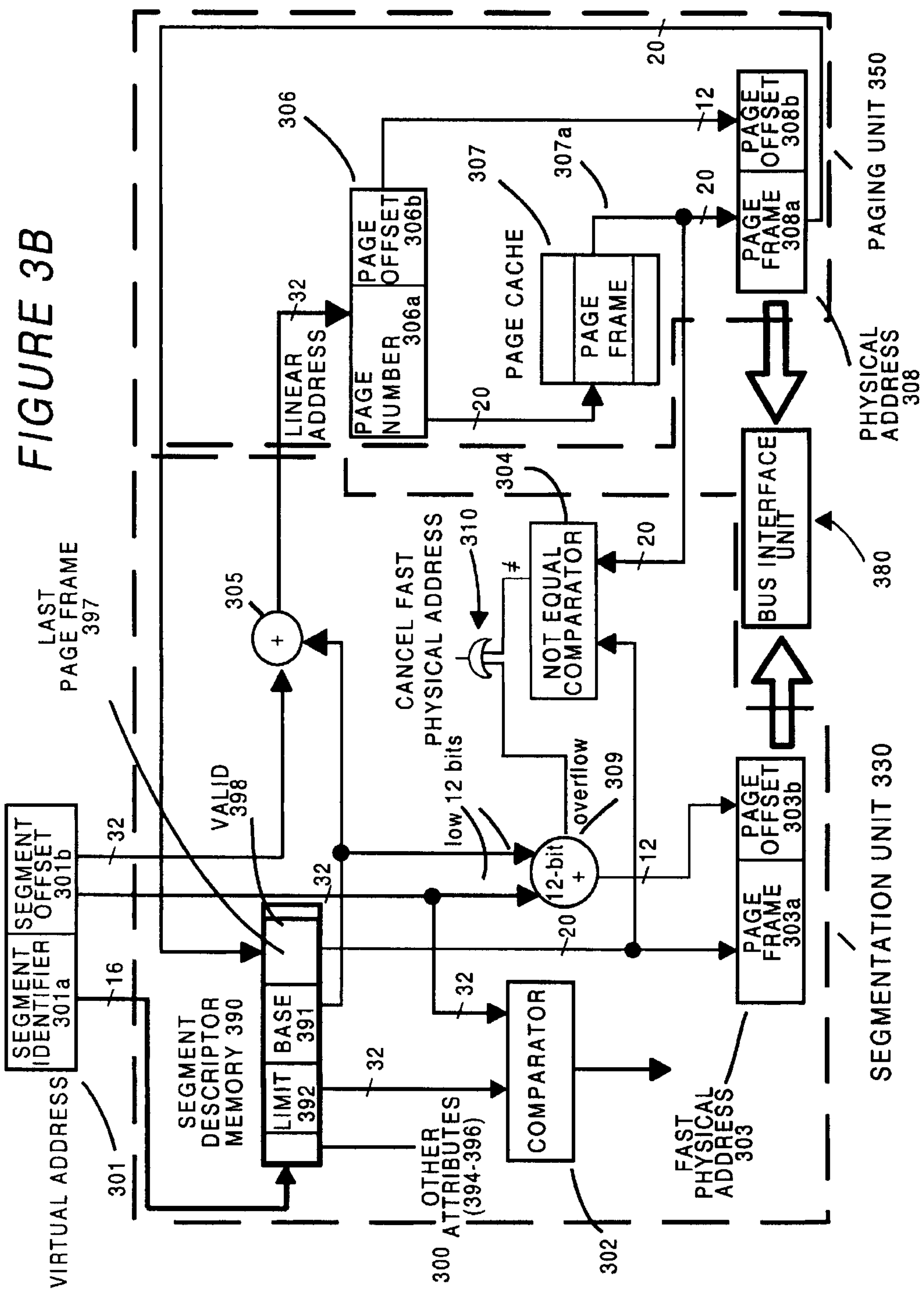
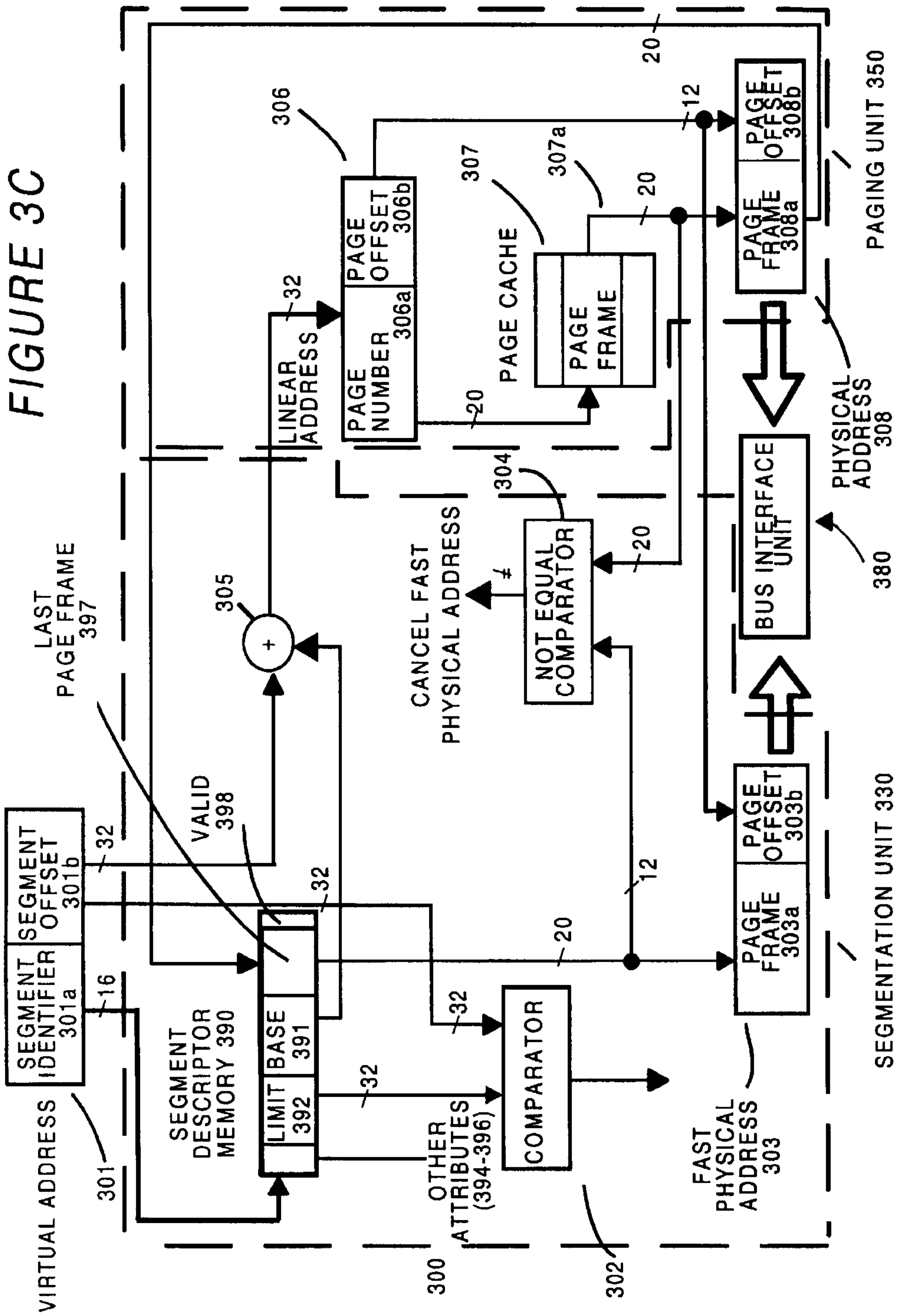

VIRTUAL ADDRESS
301
SEGMENT IDENTIFIER 301a
SEGMENT OFFSET 301b
SEGMENT DESCRIPTOR MEMORY 390
BASE 391
LIMIT 392
OTHER ATTRIBUTES (394-396)
VALID 398
LAST PAGE FRAME 397
COMPARATOR
302
300
FAST PHYSICAL ADDRESS 303
PAGE FRAME 303a
PAGE OFFSET 303b
SEGMENTATION UNIT 330
12-bit +
low 12 bits
overflow
309
CANCEL FAST PHYSICAL ADDRESS
310
NOT EQUAL COMPARATOR
304
305
LINEAR ADDRESS
PAGE NUMBER 306a
PAGE OFFSET 306b
306
PAGE CACHE
307
PAGE FRAME
307a
PAGE FRAME 308a
PAGE OFFSET 308b
PHYSICAL ADDRESS 308
PAGING UNIT 350
BUS INTERFACE UNIT
380
32
16
20
12

FIGURE 3B

FIGURE 3C



VIRTUAL ADDRESS
301
SEGMENT IDENTIFIER 301a
SEGMENT OFFSET 301b
16
32
LAST PAGE FRAME 397
VALID 398
SEGMENT DESCRIPTOR MEMORY 390
LIMIT 392
BASE 391
OTHER ATTRIBUTES (394-396)
305
+
LINEAR ADDRESS
PAGE NUMBER 306a
PAGE OFFSET 306b
306
PAGE CACHE
307
307a
PAGE FRAME
CANCEL FAST PHYSICAL ADDRESS
≠
NOT EQUAL COMPARATOR
304
COMPARATOR
302
300
20
12
PAGE FRAME 308a
PAGE OFFSET 308b
PAGING UNIT 350
PHYSICAL ADDRESS 308
BUS INTERFACE UNIT
380
PAGE FRAME 303a
PAGE OFFSET 303b
SEGMENTATION UNIT 330
FAST PHYSICAL ADDRESS 303

ADDRESS TRANSLATION MECHANISM AND METHOD IN A COMPUTER SYSTEM

This is a continuation of application Ser. No. 08/458,479 filed on Jun. 2, 1995, now U.S. Pat. No. 5,895,503. The present application is also related to a further application filed concurrently herewith entitled COMPUTER ADDRESS TRANSLATION USING FAST ADDRESS GENERATOR DURING A SEGMENTATION OPERATION PERFORMED ON A VIRTUAL ADDRESS, attorney docket no. RAB 97-001.

FIELD OF THE INVENTION

The invention relates to the field of address translation for memory management in a computer system.

BACKGROUND OF THE INVENTION

Advanced computer hardware systems operate with complex computer software programs. Computer system designers typically separate the virtual address space, the address space used by programmers in their development of software, and the physical address space, the address space used by the computer system. This separation allows programmers to think in terms of their conceptual models, and to design computer software programs without reference to specific hardware implementations. During the actual execution of programs by the computer system, however, these separate addresses must be reconciled by translating software program virtual addresses into actual physical addresses that can be accessed in a computer memory subsystem.

There are many well known approaches for address translation in the memory management mechanism of a computer system. These approaches fall into basically two major categories: those which map the smaller virtual (sometimes called logical, symbolic or user) addresses onto larger physical or real memory addresses, and those which map larger virtual addresses onto smaller physical memory. Translation mechanisms of the former category are employed typically in minicomputers in which relatively small address fields (e.g.: 16 bit addresses) are mapped onto larger real memory. Translation mechanisms of the second category are used typically in microprocessors, workstations and mainframes. Within each of these categories, segmentation only, paging only, and a combination of segmentation and paging are well known for accomplishing the translation process.

The present invention is primarily directed to address translation mechanisms where larger virtual addresses are mapped onto smaller physical addresses, and further to systems where segmentation and optional paging is employed.

In a segmentation portion of an address translation system, the address space of a user program (or programs cooperatively operating as processes or tasks), is regarded as a collection of segments which have common high-level properties, such as code, data, stack, etc. The segmented address space is referenced by a 2-tuple, known as a virtual address, consisting of the following fields: <<s>, <d>>, where <s> refers to a segment number (also called identifier or locator), and <d> refers to a displacement or offset, such as a byte displacement or offset, within the segment identified by the segment number. The virtual address <17,421>, for example, refers to the 421st byte in segment 17. The segmentation portion of the address translation mechanism, using information created by the operating system of the

computer system, translates the virtual address into a linear address in a linear address space.

In a paging portion of an address translation system, a linear (or intermediate) address space consists of a group of pages. Each page is the same size (i.e. it contains the same number of addresses in the linear space). The linear address space is mapped onto a multiple of these pages, commonly, by considering the linear address space as the 2-tuple consisting of the following fields: <<page number>, <page offset>>. The page number (or page frame number) determines which linear page is referenced. The page offset is the offset or displacement, typically a byte offset, within the selected page.

In a paged system, the real (physical) memory of a computer is conceptually divided into a number of page frames, each page frame capable of holding a single page. Individual pages in the real memory are then located by the address translation mechanism by using one or more page tables created for, and maintained by, the operating system. These page tables are a mapping from a page number to a page frame. A specific page may or may not be present in the real memory at any point in time.

Address translation mechanisms which employ both segmentation and paging are well known in the art. There are two common subcategories within this area of virtual address translation schemes: address translation in which paging is an integral part of the segmentation mechanism; and, address translation in which paging is independent from segmentation.

In prior art address translation mechanisms where paging is an integral part of the segmentation mechanism, the page translation can proceed in parallel with the segment translation since segments must start at page boundaries and are fixed at an integer number of pages. The segment number typically identifies a specific page table and the segment offset identifies a page number (through the page table) and an offset within that page. While this mechanism has the advantage of speed (since the steps can proceed in parallel) it is not flexible (each segment must start at a fixed page boundary) and is not optimal from a space perspective (e.g. an integer number of pages must be used, even when the segment may only spill over to a fraction of another page).

In prior art address translation mechanisms where paging is independent from segmentation, page translation generally cannot proceed until an intermediate, or linear, address is first calculated by the segmentation mechanism. The resultant linear address is then mapped onto a specific page number and an offset within the page by the paging mechanism. The page number identifies a page frame through a page table, and the offset identifies the offset within that page. In such mechanisms, multiple segments can be allocated into a single page, a single segment can comprise multiple pages, or a combination of the above, since segments are allowed to start on any byte boundary, and have any byte length. Thus, in these systems, while there is flexibility in terms of the segment/page relationship, this flexibility comes at a cost of decreased address translation speed.

Certain prior art mechanisms where segmentation is independent from paging allow for optional paging. The segmentation step is always applied, but the paging step is either performed or not performed as selected by the operating system. These mechanisms typically allow for backward compatibility with systems in which segmentation was present, but paging was not included.

Typical of the prior art known to the Applicant in which paging is integral to segmentation is the Multics virtual

memory, developed by Honeywell and described by the book, "The Multics System", by Elliott Organick. Typical of the prior art known to the Applicant in which optional paging is independent from segmentation is that described in U.S. Pat. No. 5,321,836 assigned to the Intel Corporation, and that described in the Honeywell DPS-8 Assembly Instructions Manual. Furthermore, U.S. Pat. No. 4,084,225 assigned to the Sperry Rand Corporation contains a detailed discussion of general segmentation and paging techniques, and presents a detailed overview of the problems of virtual address translation.

Accordingly, a key limitation of the above prior art methods and implementations where segmentation is independent from paging is that the linear address must be fully calculated by the segmentation mechanism each time before the page translation can take place for each new virtual address. Only subsequent to the linear address calculation, can page translation take place. In high performance computer systems computer systems, this typically takes two full or more machine cycles and is performed on each memory reference. This additional overhead often can reduce the overall performance of the system significantly.

SUMMARY OF THE INVENTION

An object of the present invention, therefore, is to provide the speed performance advantages of integral segmentation and paging and, at the same time, provide the space compaction and compatibility advantages of separate segmentation and paging.

A further object of the present invention is to provide a virtual address translation mechanism which architecturally provides for accelerating references to main memory in a computer system which employs segmentation, or which employs both segmentation and optional paging.

Another object of the present invention is to provide additional caching of page information in a virtual address translation scheme.

An further object of the present invention is to provide a virtual address translation mechanism which reduces the number of references required to ensure memory access.

According to the present invention, a segmentation unit converts a virtual address consisting of a segment identifier and a segment offset into a linear address. The segmentation unit includes a segment descriptor memory, which is selectable by the segment identifier. The entry pointed to by the segment identifier contains linear address information relating to the specific segment (i.e., linear address information describing the base of the segment referred to by the segment identifier, linear address information describing the limit of the segment referred to by the segment identifier, etc.) as well as physical address information pertaining to the segment—such as the page base of at least one of the pages represented by said segment.

In the above embodiment, unlike prior art systems, both segmentation and paging information are kept in the segmentation unit portion of the address translation system. The caching of this page information in the segmentation unit permits the address translation process to occur at much higher speed than in prior art systems, since the physical address information can be generated without having to perform a linear to physical address mapping in a separate paging unit.

The page base information stored in the segmentation unit is derived from the page frame known from the immediately prior in time address translation on a segment-by-segment basis. In order to complete the full physical address translation (i.e., a page frame number and page offset), the segmentation unit combines the page frame from the segment descriptor memory with the page offset field, and may store this result in a segmentation unit memory, which can be a memory table, or a register, or alternatively, it may generate the full physical address on demand.

This fast physical address generated by the segmentation unit based on the virtual address and prior page information can be used by a bus interface to access a physical location in the computer memory subsystem, even before the paging unit has completed its translation of the linear address into a page frame and page offset. Thus, fewer steps and references are required to create a memory access. Consequently, the address translation step occurs significantly faster. Since address translation occurs in a predominant number of instructions, overall system performance is improved.

The memory access is permitted to proceed to completion unless a comparison of the physical address information generated by the paging unit with the fast physical address generated by the segmentation unit shows that the page frame information of the segmentation unit is incorrect.

In alternative embodiments, the segmentation unit either generates the page offset by itself (by adding the lower portion of the segment offset and the segment base address) or receives it directly from the paging unit.

In further alternate embodiments, the incoming segment offset portion of the virtual address may be presented to the segmentation unit as components. The segmentation unit then combines these components in a typical base-plus-offset step using a conventional multiple input (typically 3-input) adder well known in the prior art.

As shown herein in the described invention, the segment descriptor memory may be a single register, a plurality of registers, a cache, or a combination of cache and register configurations.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a typical prior art virtual address translation mechanism using segmentation and independent paging.

FIG. 2A is a detailed diagram of a typical segment descriptor register of the prior art.

FIG. 2B is a detailed diagram of an embodiment of the present invention, including a portion of a segment descriptor memory used for storing physical address information;

FIG. 3A is a block diagram of an embodiment of the present invention employing segmentation and optional paging, and showing the overall structure and data paths used when paging is disabled during an address translation;

FIG. 3B is a block diagram of the embodiment of FIG. 3A showing the overall structure and data paths used when paging is enabled during an address translation;

FIG. 3C is a block diagram of another embodiment of the present invention, showing an alternative circuit for generating the fast physical address information.

DETAILED DESCRIPTION OF THE INVENTION

General Discussion of Paging & Segmentation

The present invention provides for improved virtual address translation in a computer system. The preferred embodiment employs the invention in a single-chip microprocessor system, however, it is well understood that such a virtual address translation system could be implemented in

multiple die and chip configurations without departing from the spirit or claims of the present invention.

Before embarking on a specific discussion of the present invention, however, a brief explanation of the general principles of segmentation and paging follows in order to provide additional background information, and so that the teachings of the present invention may be understood in a proper context.

Referring to FIG. 1, a typical prior art virtual address translation mechanism **100** using both segmentation and, optionally, paging in a computer system is shown. As described in this figure, a data path within the microprocessor transmits a virtual address **101**, consisting of segment identifier **101***a* and a segment offset **101***b*, to segmentation unit **130**. Segments are defined by segment descriptor entries in at least one segment descriptor table or segment descriptor segment (not shown). Segment descriptor tables are created and managed by the operating system of the computer system, and are usually located in the memory subsystem. Segment descriptor entries are utilized in the CPU of the computer system by loading them into segment descriptor register **190** or a segment descriptor cache (not shown); the segment descriptor register/cache is usually internal to the CPU, and thus more quickly accessible by the translation unit.

In the paging unit **150**, pages are defined by a page table or multiple page tables (not shown), also created and managed by the operating system; again, these tables are also typically located in a memory subsystem. All or a portion of each page table can be loaded into a page cache **107** (within the CPU, sometimes called a translation look-aside buffer) to accelerate page references.

In operation, the segmentation unit **130** first translates a virtual address to a linear address and then (except in the case when optional paging is disabled) paging unit **150** translates the linear address into a real (or physical) memory address.

Typically (as in an x86 microprocessor) the segmentation unit translates a 48-bit virtual address **101** consisting of a 16-bit segment identifier (<s>) **101***a* and a 32-bit displacement within that segment (<d>) **101***b* to a 32-bit linear (intermediate) address **106**. The 16-bit segment identifier **101***a* uniquely identifies a specific segment; this identifier is used to access an entry in a segment descriptor table (not shown). In the prior art, this segment descriptor entry contains a base address of the segment **191**, the limit of the segment **192**, and other attribute information described further below. The segment descriptor entry is usually loaded into a segment descriptor register **190**.

Using adder **105**, the segmentation unit adds the segment base **191** of the segment to the 32-bit segment offset **101***b* in the virtual address to obtain a 32-bit linear address. The 32-bit segment offset **101***b* in the virtual address is also compared against the segment limit **192**, and the type of the access is checked against the segment attributes. A fault is generated and the addressing process is aborted if the 32-bit segment offset is outside the segment limit, or if the type of the access is not allowed by the segment attributes.

The resulting linear address **106** can be treated as an offset within a linear address space; and in the commonly implemented schemes of the prior art, these offsets are frequently byte offsets. When optional paging is disabled, the linear address **106** is exactly the real or physical memory address **108**. When optional paging is enabled, the linear address is treated as a 2- or 3-tuple depending on whether the paging unit **150** utilizes one or two level page tables.

In the 2-tuple case shown in FIG. 1, which represents single level paging, the linear address, <<p>, <pd>> is divided into a page number field <p> **106***a*, and a page displacement (page offset) field within that page (<pd>) **106***b*. In the 3-tuple case (not shown) <<dp>, <p>, <pd>>, the linear address is divided into a page directory field (<dp>), a page number field <p> and a page displacement field <pd>. The page directory field indexes a page directory to locate a page table (not shown). The page number field indexes a page table to locate the page frame in real memory corresponding to the page number, and the page displacement field locates a byte within the selected page frame. Thus, paging unit **150** translates the 32-bit linear address **106** from the segmentation unit **130** to a 32-bit real (physical) address **108** using one or two level page tables using techniques which are well known in the art.

In all of the above prior art embodiments where segmentation is independent from paging, the segment descriptor table or tables of the virtual address translator are physically and logically separate from the page tables used to perform the described page translation. There is no paging information in the segment descriptor tables and, conversely, there is no segmentation information in the page tables.

This can be seen in FIG. 2A. In this figure, a typical prior art segment descriptor entry **200**, is shown as it is typically used in a segment descriptor table or segment descriptor register associated with a segmentation unit. As can be seen there, segment descriptor **200** includes information on the segment base **201**, the segment limit **202**, whether the segment is present (P) **203** in memory, the descriptor privilege level (DPL) **204**, whether the segment belongs to a user or to the system (S) **205**, and the segment type **206** (code, data, stack, etc.)

For additional discussions pertaining to the prior art in segmentation, paging, segment descriptor tables, and page tables, the reader is directed to the references U.S. Pat. Nos. 5,408,626, 5,321,836, 4,084,225, which are expressly incorporated by reference herein.

Improved Segmentation Unit Using Paging Information

As shown in the immediate prior art, the paging and segmentation units (circuits) are completely separate and independent. Since the two units perform their translation sequentially, that is, the segment translation must precede the page translation to generate the linear address, high performance computer systems, such as those employing superscalar and superpipelined techniques, can suffer performance penalties. In some cases, it is even likely that the virtual address translation could fall into the systems' "critical path". The "critical path" is a well-known characteristic of a computer system and is typically considered as the longest (in terms of gate delay) path required to complete a primitive operation of the system.

Accordingly, if the virtual address translation is in the critical path, the delays associated with this translation could be significant in overall system performance. With the recognition of this consideration, the present invention includes page information in the segment translation process. The present invention recognizes the potential performance penalty of the prior art and alleviates it by storing paging information in the segmentation unit obtained from a paging unit in previous linear-to-real address translations.

As can be seen in FIG. 2B, the present invention extends the segment descriptor entries of the prior art with a segment entry **290** having two additional fields: a LAST PAGE FRAME field **297** and a VALID field **298**. The LAST PAGE FRAME field **297** is used to hold the high-order 20 bits (i.e.:

the page frame) of the real (physical) memory address of the last physical address generated using the specified segment identifier. The VALID field **298** is a 1-bit field, and indicates whether or not the LAST PAGE FRAME field **297** is valid. The remaining fields **291–296** perform the same function as comparable fields **201–206** respectively described above in connection with FIG. **2**A.

Segment descriptor tables (not shown) can be located in a memory subsystem, using any of the techniques well-known in the art. As is also known in the art, it is possible to speed up address translation within the segmentation unit by using a small cache, such as one or more registers, or associative memory. The present invention makes use of such a cache to store segment entries **290** shown above. Unlike the prior art, however, the segment entries **290** of the present invention each contain information describing recent physical address information for the specified segment. Accordingly, this information can be used by a circuit portion of the segmentation unit to generate a new physical address without going through the linear to physical mapping process typically associated with a paging unit.

While in some instances the physical address information may change between two time-sequential virtual addresses to the same segment (and thus, a complete translation is required by both the segmentation and paging units), in the majority of cases the page frame information will remain the same. Thus, the present invention affords a significant speed advantage over the prior art, because in the majority of cases a complete virtual-linear-physical address translation is not required before a memory access is generated.

Embodiment With Segmentation & Optional Paging/Paging Disabled

Referring to FIG. **3**A, the advantage of using this new information in segment entry **290** in a segmentation unit or segmentation circuit is apparent from a review of the operation of an address translation. In this figure, a paging unit (or paging circuit) is disabled, as for example might occur only when a processor is used in a real mode of operation, rather than a protected mode of operation.

In a preferred embodiment, the present invention employs a segment descriptor memory comprising at least one, and preferably many, segment descriptor registers **390**, which are identical in every respect to the segment descriptor register described above in connection with FIG. **2**B. These segment descriptor registers are loaded from conventional segment descriptor tables or segment descriptor segments which are well known in the art. Each segment descriptor register **390** is loaded by the CPU before it can be used to reference physical memory. Segment descriptor register **390** can be loaded by the operating system or can be loaded by application programs. Certain instructions of the CPU are dedicated to loading segment descriptor registers, for example, the "LDS" instruction, "Load Pointer to DS Register". Loading by the operating system, or execution of instructions of this type, causes a base **391**, limit **392**, descriptor privilege level **394**, system/user indicator **395**, and type **396** to be loaded from segment tables or segment descriptor segments as in the prior art. The three remaining fields are present **393**, LAST PAGE FRAME **397** and VALID **398**. When a segment descriptor register **390** is loaded, present **393** is set to 1, indicating that the segment descriptor register **390** contents are present; the valid field **398** is set to 0, indicating that the last page frame number field **397** is not valid; and the LAST PAGE FRAME field **397** is not set, or may be set to 0.

After the loading of a segment descriptor register **390**, instructions of the CPU may make references to virtual

memory; if a segment descriptor register is referenced before it is loaded, as indicated by present field **393** set to 0, a fault occurs and the reference to the segment descriptor register is aborted.

As explained above, the CPU makes references to virtual memory by specifying a 48-bit virtual address, consisting of a 16-bit segment identifier **301***a* and a 32-bit segment offset **301***b*. A data path within the CPU transmits virtual address **301** to the address translation mechanism **300**.

Segment descriptor memory **390** is indexed by segment number, so each entry in this memory containing data characteristics (i.e., base, access rights, limit) of a specific segment is selectable by the segment identifier from the virtual address. Assuming this is the first reference to physical memory specifying a newly loaded segment descriptor register, since the VALID bit **398** is set to false, a prior art virtual address translation takes place. This involves, among other things, as explained earlier, various validity checks (including checking attributes **394–396**, segment limit checking using comparator **302** and potentially others), and using adder **305** to add the segment descriptor's base address **391** to the segment offset **301***b* to calculate a linear address **306**.

While the implementation in the embodiment of FIG. **3**A shows the addition of the base address **391** to the segment offset **301***b* using adder **305** to generate the linear address **306**, it will be understood by those skilled in the art that this specific implementation of the virtual to linear address translation is not the only implementation of the present invention. In other implementations, the segment offset **301***b* might consist of one or more separate components. Different combinations of one or more of these components might be combined using well known techniques to form a linear address, such as one utilizing a three-input adder. The use of these components is discussed, for example, in U.S. Pat. No. 5,408,626, and that description is incorporated by reference herein.

As is well known, in this embodiment where paging is disabled, linear address **306** is also a physical address which can be used as the physical address **308**. Memory access control operations are not shown explicitly since they are only ancillary to the present invention, and are well described in the prior art. In general, however, a bus interface unit **380** is typically responsible for interactions with the real (physical) memory subsystem. The memory subsystem of a computer system employing the present invention preferably has timing and address and data bus transaction details which are desirably isolated from the CPU and address translation mechanism. The bus interface unit **380** is responsible for this isolation, and can be one of many conventional bus interface units of the prior art.

In the present invention, bus interface unit **380** receives the real memory address **308** from address translation mechanism **300** and coordinates with the real memory subsystem to provide data, in the case of a memory read request, or to store data, in the case of a memory write request. The real memory subsystem may comprise a hierarchy of real memory devices, such as a combination of data caches and dynamic RAM, and may have timing dependencies and characteristics which are isolated from the CPU and address translation mechanism **300** of the computer system by the bus interface unit **380**.

Simultaneous with the first memory reference using the calculated physical address **308**, the LAST PAGE FRAME field of the selected segment descriptor register **390** is loaded with the high-order 20 bits of the physical address,

i.e.: the physical page frame, and the VALID bit is set to indicate a valid state. This paging information will now be used in a next virtual address translation.

Accordingly, when a next, new virtual address **301** is to be translated, the entry selected from segment descriptor memory **390** will likely contain the correct physical frame page number (in the LAST PAGE FRAME field **397**). Thus, in most cases, the base physical address in memory for the next, new referenced virtual address will also be known from a previous translation.

The first step of the virtual address translation, therefore, is to determine if a FAST PHYSICAL ADDRESS **303** can be used to begin a fast physical memory reference. Adder **309**, a 12-bit adder, adds the low-order 12 bits of the segment offset **301***b* of virtual address **301** to the low-order 12-bits of base **391** of the segment entry in segment descriptor register **390** referenced by the segment identifier **301***a*. This addition results in a page offset **303***b*. In parallel with adder **309**, 32-bit adder **305** begins a full 32-bit add of segment base **391** and segment offset **301***b*, to begin producing the linear address; however, this full 32-bit add will obviously require more time. In the preferred embodiment, adder **309** is a separate 12-bit adder; however, it should be noted that adder **309** also could be implemented as the low order 12-bits of 32-bit adder **305**.

Simultaneous with the beginning of these two operations, VALID bit **398** is inspected. If VALID bit **398** is set to 1, as soon as 12-bit adder **309** has completed, 20-bit LAST PAGE FRAME **397** is concatenated with the result of adder **309** to produce FAST PHYSICAL ADDRESS **303**, consisting of a page frame number **303***a*, and page offset **303***b*. FAST PHYSICAL ADDRESS **303** then can be used to tentatively begin a reference to the physical memory. It should be understood that the FAST PHYSICAL ADDRESS **303** transmitted to bus interface unit **380** could also be stored in a register or other suitable memory storage within the CPU.

In parallel with the fast memory reference, limit field **392** is compared to the segment offset **301***b* of the virtual address by comparator **302**. If the offset in the virtual address is greater than the limit, a limit fault is generated, and virtual address translation is aborted.

Also in parallel with the fast memory reference, adder **305** completes the addition of base **391** to the segment offset field **301***b* of virtual address to produce linear address (in this case physical address also) **306**. When this calculation is completed, the page frame number **308***a* of physical address **308** is compared to LAST PAGE FRAME **397** by Not Equal Comparator **304**. If page frame **308***a* is unequal to the LAST PAGE FRAME **397**, or if 12-bit Adder **309** overflowed (as indicated by a logic "1" at OR gate **310**), the fast memory reference is canceled, and the linear address **306**, which is equal to the physical address **308**, is used to begin a normal memory reference. If page frame **308***a* is equal to LAST PAGE FRAME **397**, and 12-bit Adder **309** did not overflow (the combination indicated by a logic "0" at the output of OR gate **310**), the fast memory reference is allowed to fully proceed to completion.

After any fast memory reference which is cancelled by the CANCEL FAST PHYSICAL ADDRESS signal output of OR gate **310**, page frame **308***a* is loaded into the LAST PAGE FRAME **397** in the segment descriptor memory **390** for subsequent memory references.

Depending on the particular design desired, it should also be noted that writes to the memory, or reads which cause faults using FAST PHYSICAL ADDRESS **303** may be pended since the FAST PHYSICAL ADDRESS **303** may prove to be invalid.

Accordingly, it can be seen that the parallel physical address calculation undertaken by the improved segmentation unit of the present invention generates a faster physical memory access than possible with prior art systems.

Embodiment With Segmentation & Paging/Paging Enabled

The present invention can also be used with address translation units using paging enabled, as can be seen in the embodiments of FIGS. **3**B and **3**C.

In the embodiment of FIG. **3**B, the same segmentation unit structure **300** as that shown in FIG. **3**A is used, and the operation of segmentation unit **300** is identical to that already explained above. As before, segment descriptor memory (registers) **390** are loaded from conventional segment descriptor tables or segment descriptor segments, using one or more of the procedures described above. First, the base **391** limit **392** descriptor privilege level **394**, system/user indicator **395**, and type **396** are loaded from segment tables or segment descriptor segments as explained earlier. When segment descriptor register **390** is loaded, present **393** is set to 1, indicating that the segment descriptor register **390** contents are present; the valid field **398** is set to 0, indicating that the last page frame number field **397** is not valid; and the LAST PAGE FRAME field **397** is not set, or may be set to 0.

As explained above, after the loading of a segment descriptor register **390**, instructions of the CPU may make references to virtual memory; if a segment descriptor register is referenced before it is loaded, as indicated by present field **393** set to 0, a fault occurs and the reference to the segment descriptor register is aborted.

As further explained above, the 48 bit virtual address **301** (consisting of a 16 bit segment identifier **301***a* and a 32 bit segment offset **301***b*) is transmitted by a data path to segmentation unit **300**, and an index into segment descriptor memory **390** is performed to locate the specific segment descriptor for the segment pointed to by segment identifier **301***a*. Assuming this is the first reference to physical memory specifying a newly loaded segment descriptor register, since the VALID bit is set to false, a prior art virtual address translation takes place. This involves, among other things, as explained earlier, various validity checks (including checking attributes **394**–**396**, segment limit checking using comparator **302** and potentially others), and using adder **305** to add the segment descriptor's base address **391** to the segment offset **301***b* to calculate a linear address **306**.

As is well known, in this configuration where paging is enabled, linear address **306** must undergo a further translation by paging unit **350** to obtain the physical address **308** in the memory subsystem. In the preferred embodiment of the invention, looking first at FIG. **3**B, the output of adder **305** will be a 32-bit linear address, corresponding to a 20-bit page number **306***a* and a 12-bit page offset **306***b*. Typically, the page number **306***a* is then indexed into a page descriptor table (not shown) to locate the appropriate page frame base physical address in memory. These page descriptor tables are set up by the operating system of the CPU using methods and structures well known in the art, and they contain, among other things, the base physical address of each page frame, access attributes, etc.

However, in most systems, including the present invention, a page cache **307** is used in order to hold the physical base addresses of the most recently used page frames. This cache can take the form of a table, associative cache, or other suitable high speed structure well known in

the art. Thus, page number **306***a* is used to access page data (including physical base addresses for page frames) in an entry in page cache **307**.

If page cache **307** hits, two things happen: first, a 20-bit PAGE FRAME **307***a* (the page frame in physical memory) replaces the high-order 20 bits (page number **306***a*) of the linear address **306**, and, when concatenated with the page offset **306***b* results in a real (physical) address **308**, which is used to perform a memory access through bus interface unit **380** along the lines explained above. Second, newly generated page frame **308***a* is also stored in segment descriptor memory **390** in the selected LAST PAGE FRAME field **397** to be used for a fast access in the next address translation. When LAST PAGE FRAME field **397** is stored, selected VALID bit **398** is set to 1 to indicate that LAST PAGE FRAME **397** is valid for use.

In the event of a page cache miss, the appropriate page frame number **308***a* is located (using standard, well-known techniques) to generate physical address **308**, and is also loaded into segment descriptor memory **390** in the selected LAST PAGE FRAME field **397**. The selected VALID bit **398** is also set to indicate a valid state. Thus, there is paging information in the segmentation unit that will now be used in the next virtual address translation.

When a next, new virtual address **301** is to be translated, the segment identifier **301***a* will likely be the same as that of a previously translated virtual address, and the entry selected from segment descriptor memory **390** will also likely contain the correct physical frame (in LAST PAGE FRAME field **397**) from the previous translation. As with the above embodiment, one or more registers, or a cache may be used for the segment descriptor memory **390**.

The first step then determines if a FAST PHYSICAL ADDRESS **303** can be used to begin a fast physical memory reference. Adder **309**, a 12-bit adder, adds the low-order 12 bits of the segment offset **301***b* of virtual address **301** to the low-order 12-bits of base **391** of the segment entry in segment descriptor register **390** referenced by the segment identifier **301***a*. This addition results in a page offset **303***b*. In parallel with adder **309**, 32-bit adder **305** begins a full 32-bit add of segment base **301** and segment offset **301***b*, to begin producing the linear address; however, this full 32-bit add will obviously require more time. In the preferred embodiment, adder **309** is a separate 12-bit adder, however, it should be noted that adder **309** also could be implemented as the low order 12-bits of 32-bit adder **305**.

Simultaneous with these beginning of these two operations, VALID bit **398** is inspected. If VALID bit **398** is set to 1, as soon as 12-bit adder **309** has completed, 20-bit LAST PAGE FRAME **397** is concatenated with the result of adder **309** to produce FAST PHYSICAL ADDRESS **303**, consisting of a page frame number **303***a*, and page offset **303***b*. FAST PHYSICAL ADDRESS **303** then can be used to tentatively begin a reference to the physical memory. Again, it should be understood that the FAST PHYSICAL ADDRESS **303** transmitted to bus interface unit **380** could also be stored in a register or other suitable memory storage within the CPU.

As before, limit field **302** is compared to the segment offset **301***b* of the virtual address by comparator **302**. If the offset in the virtual address is greater than the limit, a limit fault is generated, and virtual address translation is aborted.

This new virtual address is also translated by paging unit **350** in the same manner as was done for the previous virtual address. If page cache **307** hits based on the page number **306***a*, two things happen: first, a 20-bit PAGE FRAME **307***a* (the page frame in physical memory) replaces the high-order 20 bits (page number **306***a*) of the linear address **306**, and, when concatenated with the page offset **306***b* results in a physical address **308**. This real address may or may not be used, depending on the result of the following: in parallel with the aforementioned concatenation, the PAGE FRAME **307***a*, is compared to LAST PAGE FRAME **397** from the segment descriptor memory **390** by Not Equal Comparator **304**. The result of Not Equal Comparator (that is, the Not Equal condition) is logically ORed with the overflow of 12-bit adder **309** by OR gate **310**. If the output of OR gate **310** is true (i.e. CANCEL FAST PHYSICAL ADDRESS is equal to binary one), or if PAGE CACHE **307** indicates a miss condition, the fast memory reference previously begun is canceled, since the real memory reference started is an invalid reference. Otherwise, the fast memory reference started is allowed to fully proceed to completion, since it is a valid real memory reference.

If CANCEL FAST PHYSICAL ADDRESS is logical true, it can be true for one of two, or both reasons. In the case that Or gate **310** is true, but page cache **307** indicates a hit condition, physical address **308** is instead used to start a normal memory reference. This situation is indicative of a situation where LAST PAGE FRAME **397** is different from the page frame **308***a* of the current reference.

In the case that page cache **307** did not indicate a hit, a page table reference through the page descriptor table is required and virtual address translation proceeds as in the prior art. The page frame **308***a* information is again stored in the LAST PAGE FRAME field **397** in the segment descriptor memory **390** for the next translation.

Also, after any fast memory reference which is canceled by the CANCEL FAST PHYSICAL ADDRESS signal output of OR gate **310**, page frame number **308***a* is loaded into the LAST PAGE FRAME **397** in the segment descriptor memory **390** for subsequent memory references.

Depending on the particular design desired, it should also be noted that in this embodiment also, writes to the memory, or reads which cause faults using FAST PHYSICAL ADDRESS **303** may be pended since the FAST PHYSICAL ADDRESS **303** may prove to be invalid.

The alternative embodiment shown in FIG. **3**C is identical in structure and operation to the embodiment of FIG. **3**B, with the exception that the 12-bit Adder **309** is not employed. In this embodiment, the segmentation unit **330** does not create the lower portion (page offset **303***a*) of the fast physical address in this manner. Instead, the page offset **306***a* resulting from 32-bit adder **305** is used.

It can be seen that the present invention has particular relevance to computers using sequential type of segmentation and paging translation, such as the X86 family of processors produced by the Intel Corporation (including the Intel 80386, Intel 80486 and the Intel Pentium Processor), other X86 processors manufactured by the NexGen Corporation, Advanced Micro Devices, Texas Instruments, International Business Machines, Cyrix Corporation, and certain prior art computers made by Honeywell. These processors are provided by way of example, only, and it will be understood by those skilled in the art that the present invention has special applicability to any computer system where software executing on the processors is characterized by dynamic execution of instructions in programs in such a way that the virtual addresses are generally logically and physically located near previous virtual addresses.

The present invention recognizes this characteristic, employing acceleration techniques for translating virtual to

real addresses. In particular, the present invention utilizes any of the commonly known storage structures (specific examples include high speed registers and/or caches) to store previous address translation information, and to make this previous address translation information available to the system whenever the next subsequent reference relies on the same information. In this way, the system can utilize the previously stored information from the high speed storage to begin real memory references, rather than be forced to execute a more time consuming translation of this same information, as was typically done in the prior art.

As will be apparent to those skilled in the art, there are other specific circuits and structures beyond and/or in addition to those explicitly described herein which will serve to implement the translation mechanism of the present invention. Finally, although the above description enables the specific embodiment described herein, these specifics are not intended to restrict the invention, which should only be limited as defined by the following claims.

I claim:

1. A system for performing address translations usable by a processor employing both segmentation and optional independent paging the system comprising:

means for generating an actual physical address from a virtual address in a time period T, said virtual address having both a segment identifier and a segment offset by calculating a linear address based on said entire virtual address, and by calculating said actual physical address based on said calculated linear address; and

a fast physical address generator for generating a fast physical address related to said virtual address in a time<T.

2. The system of claim **1**, wherein the fast physical address can be used for generating a memory access faster than a memory access based on said actual physical address.

3. The system of claim **2**, including a cancellation circuit for cancelling the memory access if the fast physical address and actual physical address are different.

4. The system of claim **1**, wherein the fast physical address is generated based on a combination of physical address information from a different virtual address, and partial linear address information relating to said virtual address.

5. The circuit of claim **1**, wherein the fast physical address is generated before said calculated linear address.

6. A system for performing address translations using a first operation to convert a first virtual address having both a segment identifier portion and a segment offset portion to a first linear address, the first linear address being based on all portions of the virtual address and a second operation to convert said first linear address to a first physical address, said system further including:

a tentative physical address generator for generating a tentative physical address related to said first virtual address;

wherein the tentative physical address can be generated before said second operation has completed converting said first linear address.

7. The system of claim **6**, wherein the tentative physical address can be used for generating a memory access which is faster than a memory access resulting from said first physical address.

8. The system of claim **7**, including a cancellation circuit for cancelling the memory access if the tentative physical address and first physical address are different.

9. The system of claim **6**, wherein the tentative physical address is generated based on a combination of prior physical address information and partial linear address information relating to said first virtual address.

10. The circuit of claim **6**, wherein the tentative physical address is generated before said first operation has completed converting said first virtual address into said first linear address.

11. The circuit of claim **6**, wherein said first virtual address is partially converted to a linear address by the fast physical address circuit and is combined with physical address information relating to a prior virtual address to generate the tentative physical address.

12. A computer system which performs address translations using a first operation to convert virtual addresses having both a segment identifier portion and a segment offset portion to linear addresses, such that both the segment identifier and segment offset portions of the virtual addresses are used for converting said linear addresses and a second operation to convert said linear addresses to physical addresses, said system further including:

a fast physical address generator for generating fast physical addresses related to said virtual addresses;

wherein the fast physical addresses can be generated while or before said virtual addresses are converted in said first operation into said linear addresses.

13. The system of claim **12**, wherein the fast physical addresses can be used for generating memory accesses faster than memory accesses resulting from said calculated physical addresses.

14. The system of claim **13**, including a cancellation circuit for cancelling the memory accesses if the fast physical addresses and calculated physical addresses are different.

15. The system of claim **12**, wherein the fast physical addresses are generated based on a combination of physical address information and partial linear address information relating to said virtual addresses.

16. The system of claim **12**, wherein said virtual addresses are partially converted to linear addresses by the fast physical address circuit and are combined with physical address information relating to prior virtual addresses to generate the tentative physical addresses.

17. A system for performing address translations comprising:

a virtual to linear address converter circuit for generating a calculated linear address based on a virtual address, said virtual address having both a segment identifier and a segment offset, and said calculated linear address being based on all of said virtual address; and

a linear to physical address converter circuit for generating a calculated physical address based on the calculated linear address, the calculated physical address including a calculated page frame and a calculated page offset; and

a fast physical address circuit for generating a fast physical address including a fast page frame and a fast page offset;

wherein a memory reference can be generated based on the fast physical address;

further wherein the fast physical address is based on linear address information relating to the virtual address and physical address information relating to a prior virtual address.

18. A system for performing address translations comprising:

a virtual to linear address converter circuit for generating a calculated linear address based on a virtual address, said virtual address having both a segment identifier

and a segment offset, and said calculated linear address being based on all of said virtual address; and

a linear to physical address converter circuit for generating a calculated physical address based on the calculated linear address, the calculated physical address including a calculated page frame and a calculated page offset; and

a fast physical address circuit for generating a fast physical address including a fast page frame and a fast page offset,

wherein a memory reference can be generated based on the fast physical address;

further wherein the virtual address is partially converted to a linear address by the fast physical address circuit and is combined with physical address information relating to a prior virtual address to generate the tentative physical address.

19. A system for performing address translations using a first operation to convert a first virtual address having both a segment identifier portion and a segment offset portion to a first linear address, such that all portions of the virtual address are considered when converting said virtual address into the first linear address and a second operation to convert said first linear address to a first physical address, the system further including:

an address translation memory, accessible by said system while said first operation is converting said first virtual address, and capable of storing prior physical address information generated during a prior address translation by said second operation based on a prior virtual address;

wherein a fast physical address can be generated based on the prior physical address information and said first linear address before said second operation has completed converting said first linear address to the first physical address.

20. The system of claim 19, wherein the fast physical address can be used for an accelerated memory access which is faster than a memory access resulting from said first physical address.

21. The system of claim 20, including a cancellation circuit for cancelling the fast memory access if the fast physical address and first physical address are different.

22. The system of claim 19, wherein the fast physical address is comprised of:

(iii) a page frame portion based on the prior physical address information; and

(iv) a page offset portion based on the result of converting said first virtual address to a first linear address.

23. A computer system using segmentation and optional independent paging for performing address translations comprising:

an address translation memory capable of storing:

(i) a portion of a physical address corresponding to a stored page frame; and

(ii) segment base information relating to a virtual address; and

a virtual to linear address converter circuit for generating a calculated linear address based on combining segment offset portion of the virtual address and the segment base, wherein all of said virtual address is used for generating the calculated linear address; and

a linear to physical address converter circuit for receiving and generating a calculated physical address based on the calculated linear address, the calculated physical address including a first page frame and a first page offset; and

a fast physical address circuit for generating a fast physical address comprised of the stored page frame combined with a fast page offset portion derived from the segment base and the virtual address;

wherein the fast physical address is calculated prior to the generation of said calculated physical address.

24. The system of claim 23, wherein the fast physical address can be used for generating a fast memory access which is generated more quickly than a memory access resulting from said first physical address.

25. The system of claim 23, including a cancellation circuit for cancelling the fast memory access if the fast physical address and first physical address are different.

26. The circuit of claim 23, wherein the fast physical address is generated prior to the generation of the first linear address.

27. The system of claim 23, wherein the stored page frame is generated in a prior address translation based on a prior virtual address.

28. A method of performing a translation of a virtual address in a computer system using segmentation and optional independent paging, said method including the steps of:

(a) calculating a fast physical address related to said virtual address; and

(b) calculating a linear address based on said virtual address, said linear address being based on both a segment identifier and segment offset portion of said virtual address; and

(c) calculating an actual physical address based on the linear address;

wherein step (a) is completed prior to the completion of step (c), and the fast physical address can be used to initiate a fast memory reference.

29. The method of claim 28, further including a step (d): cancelling the memory access if the fast physical address and actual physical address are different.

30. The method of claim 28, wherein the fast physical address is generated based on a combination of physical address information from a different virtual address, and partial linear address information relating to said virtual address.

31. The method of claim 28, wherein step (a) is completed prior to the completion of step (b).

32. A method of generating memory references based on virtual addresses in a computer system, said computer system using segmentation and optional independent paging, the method including the steps of:

(a) generating tentative memory references based on said virtual addresses; and

(b) converting said virtual addresses to linear addresses during a segmentation operation, said linear addresses being based on translating all portions of said virtual address; and

(c) converting said linear addresses to physical addresses during a paging operation, so that actual memory references can be made based on said physical addresses;

wherein the tentative memory reference can be generated while said virtual addresses are being converted in said first operation into said linear addresses.

33. The method of claim 32, further including a step (d): cancelling the tentative memory reference if the tentative memory reference and actual memory reference are different.

34. The method of claim 32, wherein the tentative memory reference is generated based on a combination of

physical address information and partial linear address information relating to said virtual addresses.

35. The method of claim **32**, wherein step (a) is completed prior to the completion of step (b).

36. A method of generating a fast memory reference using a fast physical address derived from a virtual address having both a segment identifier and a segment offset in a computer system employing both segmentation and optional independent paging, the method including the steps of:

(a) converting a portion of said virtual address into a partial linear address; and

(b) combining the partial linear address with physical address information obtained from a prior memory reference to generate said fast physical address;

(c) generating a memory reference based on the fast physical address;

(d) converting said virtual address into an actual physical address during which time a linear address is also calculated based on both the <segment id> and <segment offset> of said virtual address;

(e) cancelling the memory reference if the fast physical address and actual physical address are different.

37. The method of claim **36**, wherein the fast physical address is generated prior to the generation of the linear address.

38. The method of claim **36**, wherein the fast physical address is used to generate a fast memory access prior to the generation of the linear address.

39. A method of generating physical addresses from virtual addresses in a computer system employing both segmentation and optional independent paging, the method including the steps of:

(a) generating a first calculated linear address based on a first virtual address in a first operation, said linear addresses being based on translating all portions of said first virtual address; and

(b) generating a fast physical address in a second operation, the fast physical address including linear address information relating to said first virtual address and portions of physical address information relating to said first virtual address; and

(c) generating a first calculated physical address in a third operation based on the first calculated linear address;

wherein the fast physical address is generated prior to the generation of the first calculated physical address.

40. The method of claim **39**, wherein the fast physical address is used to generate a tentative memory access prior to the generation of the first calculated physical address.

41. The method of claim **40**, including a step (d): cancelling the tentative memory access if the fast physical address and first calculated physical address are different.

42. The method of claim **39**, further including a step (e): generating a memory access request based on the first calculated physical address; and (f) storing physical address information relating to the first calculated physical address for use in a later address translation.

43. The method of claim **39**, wherein the first and second operations overlap in time, and the fast physical address is generated prior to the generation of the first calculated linear address.

44. A system for performing memory references in a processor which employs both segmentation and optional independent paging during an address translation, said system comprising:

means for performing an address translation by generating a first physical address from a first virtual address by first calculating a first linear address based on both a first segment identifier and first offset associated with the first virtual address, such that all of said first virtual address is translated, and then calculating the first physical address based on the first calculated linear address; and

a fast physical memory access circuit for generating a fast memory reference, which fast memory reference is based on physical address information from said means for performing an address translation;

a bus interface circuit for initiating a fast memory access to a memory subsystem based on said fast memory reference.

45. The system of claim **42**, further including a comparator for determining whether said fast memory reference can be used for a fast memory access.

46. The system of claim **45**, further including a cancellation circuit for canceling said fast memory access.

47. The system of claim **46**, wherein the system performs an actual memory reference after said fast memory reference is cancelled.

48. A method for performing memory accesses between a processor and a memory, said processor having an address translation mechanism that employs segmentation and optional independent paging, the method comprising the steps of:

generating computed physical addresses by converting virtual addresses having a segment identifier and a segment offset into linear addresses, such that all portions of said virtual addresses are translated, and then converting said linear addresses into a physical addresses;

generating a speculative physical address based on one of said computed physical addresses;

initiating a speculative memory access based on said speculative physical address.

49. The method of claim **48**, further including a step of initiating an actual memory access based on a physical address which has been computed during separate segmentation and paging operations.

50. The method of claim **49**, wherein said speculative memory access is completed unless canceled in favor of an actual memory access.

51. A system for performing a first and a second address translation of first and second virtual addresses respectively, the system comprising:

a virtual to linear address converter circuit for generating a first calculated linear address based on translating all portions of the first virtual address including a segment identifier and a segment offset; and

a linear to physical address converter circuit for completing the first address translation by generating a first calculated physical address based on said first calculated linear address, said first calculated physical address including a first calculated page frame and a first calculated page offset; and

wherein the system uses information from the first address translation during the second address translation so that the second address translation can be performed faster than the first address translation.

52. The system of claim **51**, further including a comparator for determining whether said second address translation can be used for a memory access.

53. The system of claim **51**, wherein said second address translation is based on a combination of partial linear address information relating to said second virtual address and physical address information from a different virtual address.

54. The system of claim **51**, wherein the system also calculates an actual second physical address from said second virtual address, by calculating a second linear address based on a second segment identifier and second offset associated with said second virtual address, and calculating said second physical address based on said second calculated linear address.

55. The system of claim **54**, wherein at least a portion of said actual second physical address is compared with a corresponding portion of said second physical address from said fast physical address generator, and when said portions are not equal, said actual second physical address is used for a memory access.

56. The system of claim **51**, further including a register for storing address information pertaining to the first virtual address for use during said translation of said second virtual address.

57. A circuit for performing fast translations of virtual addresses to physical addresses in a computer system which uses both segmentation and optional independent paging, the circuit including:

an address generator for performing a first address translation of a first virtual address having an associated first segment identifier and a first offset, said first translation including converting all of said virtual address into a first linear address;

said address generator also performing a fast address translation of a second virtual address having an associated second segment identifier and a second offset, said fast address translation occurring without converting all of said second virtual address into a second linear address;

wherein said address generator uses information from the first address translation during the fast address translation so that said translation of said second virtual address takes less time than said first address translation.

58. The system of claim **57**, further including a comparator for determining whether said fast address translation can be used for a memory access.

59. The system of claim **57**, wherein said fast address translation is achieved based on a combination of partial linear address information relating to said second virtual address and physical address information from said first virtual address.

60. The system of claim **57**, wherein the address generator also performs a calculated translation to calculate an actual second physical address from said second virtual address, by calculating a second linear address based on said second segment identifier and second offset associated with said second virtual address, and calculating said second physical address based on said second calculated linear address.

61. The system of claim **60**, wherein at least a portion of said actual second physical address is compared with a corresponding portion of said second physical address from said fast physical address generator, and when such portions are not equal, said actual second physical address is used for a memory access.

62. The system of claim **57**, further including a register for storing address information pertaining to the first virtual address for use during said translation of said second virtual address.

63. A method of translating virtual addresses in a computer system that uses both segmentation and optional independent paging, the method including the steps of:

(a) generating a first calculated physical address based on a first virtual address in a first operation, said first virtual address including a first segment identifier and a first offset and wherein said first operation converts all of said virtual address into a first linear address; and

(b) generating a second fast physical address in a second operation based on a second virtual address, said second virtual address including a second segment identifier and a second offset, and said second fast physical address being generated based on information obtained during said first operation, and without converting all of said second virtual address into a second linear address:

wherein said second operation is performed faster than said first operation.

64. The method of claim **63**, further including a step of determining whether a memory access can be made using said second fast physical address.

65. The method of claim **63**, wherein during step (b) said second physical address is generated based on a combination of partial linear address information relating to said second virtual address and physical address information from said first virtual address.

66. The method of claim **63**, further including a step (c): generating an actual second physical address from said second virtual address during a third operation, by calculating a second linear address based on said second segment identifier and second offset associated with said second virtual address, and calculating said second physical address based on said second calculated linear address.

67. The method system of claim **66**, further including step (d): comparing at least a portion of said actual second physical address with a corresponding portion of said second physical address from said fast physical address generator, and when such portions are not equal, using said actual second physical address for a memory access.

68. The system of claim **63**, further including a step of storing address information pertaining to the first virtual address in a register during said first operation for use during said second operation.

69. A method of performing address translations in a computer system that uses both segmentation and optional independent paging, the method including the steps of:

(a) performing a first address translation by translating a first virtual address into a first physical address by: (i) first calculating a first linear address based on a first segment identifier and first offset associated with said first virtual address wherein all of said virtual address is translated; and (ii) calculating said first physical address based on said first calculated linear address and

(b) performing a second address translation using information obtained during said first address translation to translate a second virtual address into a second physical address, said second physical address being obtained without converting all of said second virtual address into a second linear address;

wherein said second translation can be achieved in less time than said first translation.

70. The method of claim **69**, further including a step of determining whether a memory access can be made using said second physical address.

71. The method of claim **69**, wherein during step (b) said second physical address is generated based on a combination of partial linear address information relating to said second virtual address and physical address information from said first virtual address.

72. The method of claim **69**, further including a step (c): generating an actual second physical address from said second virtual address, by calculating a second linear

address based on said second segment identifier and second offset associated with said second virtual address, and calculating said second physical address based on said second calculated linear address.

73. The method system of claim **72**, further including step (d): comparing at least a portion of said actual second physical address with a corresponding portion of said second physical address from said fast physical address generator, and when such portions are not equal, using said actual second physical address for a memory access.

74. The system of claim **69**, further including a step of storing address information pertaining to the first virtual address in a register for use during said second address translation.

* * * * *

EXHIBIT D

US006430668B2

(12) **United States Patent**
Belgard

(10) **Patent No.: US 6,430,668 B2**
(45) **Date of Patent: Aug. 6, 2002**

(54) **SPECULATIVE ADDRESS TRANSLATION FOR PROCESSOR USING SEGMENTATION AND OPTICAL PAGING**

(75) Inventor: **Richard Belgard**, Saratoga, CA (US)

(73) Assignee: **Transmeta Corporation**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/757,439**

(22) Filed: **Jan. 10, 2001**

Related U.S. Application Data

(63) Continuation of application No. 08/458,479, filed on Jun. 2, 1995, now Pat. No. 5,895,503, and a continuation of application No. 09/905,410, filed on Aug. 4, 1997, now Pat. No. 5,960,466, and a continuation of application No. 08/905, 356, filed on Aug. 4, 1997, now Pat. No. 6,226,733.

(51) **Int. Cl.**[7] **G06F 12/10**
(52) **U.S. Cl.** **711/202**; 711/201; 711/203; 711/206; 711/208; 711/209; 711/217; 711/218; 711/219; 711/220
(58) **Field of Search** 711/201–206, 208, 711/209, 217–220

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,084,225 A		4/1978	Anderson et al.	711/206
4,400,774 A		8/1983	Toy	711/3
5,165,028 A		11/1992	Zulian	711/3
5,321,836 A	*	6/1994	Crawford et al.	711/206
5,335,333 A		8/1994	Hinton et al.	711/207
5,423,014 A		6/1995	Hinton et al.	711/3
5,617,554 A	*	4/1997	Alpert et al.	711/208
5,895,503 A	*	4/1999	Belgard	711/202
5,960,466 A	*	9/1999	Belgard	711/213
6,226,733 B1	*	5/2001	Belgard	711/213

FOREIGN PATENT DOCUMENTS

EP 0668665 8/1995

OTHER PUBLICATIONS

U5S 486 Green CPU, United Microelectronics Corporation, 1994–95, pp. 3–1 to 3–26.

Intel Microprocesors, vol. 1, Intel Corporation, 1993, pp. 2–229 to 2–287.

Computer Architecture A Quantitative Approach, Hennessey and Patterson, pp. 432–497 (1990).

Hua, A. Hunt, L. Liu, J–K Peir, D. Pruett, and J. Temple, "Early Resolution of Address Translation in Cache Design," Proc. of Int'l Conf. on Computer Designs, Oct. 1990, pp. 408–412.

DPS–8 Assembly Instructions, Honeywell Corporation, Apr., 1980, Chapter 3 and 5.

The Multics System, Elliot Organick, 1972, pp. 6–7, 38–51.

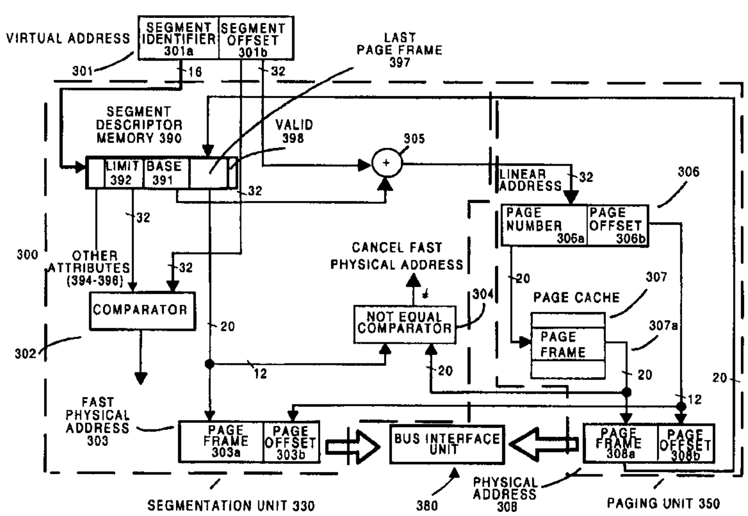
* cited by examiner

Primary Examiner—Than Nguyen
(74) *Attorney, Agent, or Firm*—J. Nicholas Gross

(57) **ABSTRACT**

An improved address translation method and mechanism for memory management in a computer system is disclosed. A segmentation mechanism employing segment registers maps virtual addresses into a linear address space. A paging mechanism optionally maps linear addresses into physical or real addresses. Independent protection of address spaces is provided at each level. Information about the state of real memory pages is kept in segment registers or a segment register cache potentially enabling real memory access to occur simultaneously with address calculation, thereby increasing performance of the computer system.

26 Claims, 5 Drawing Sheets



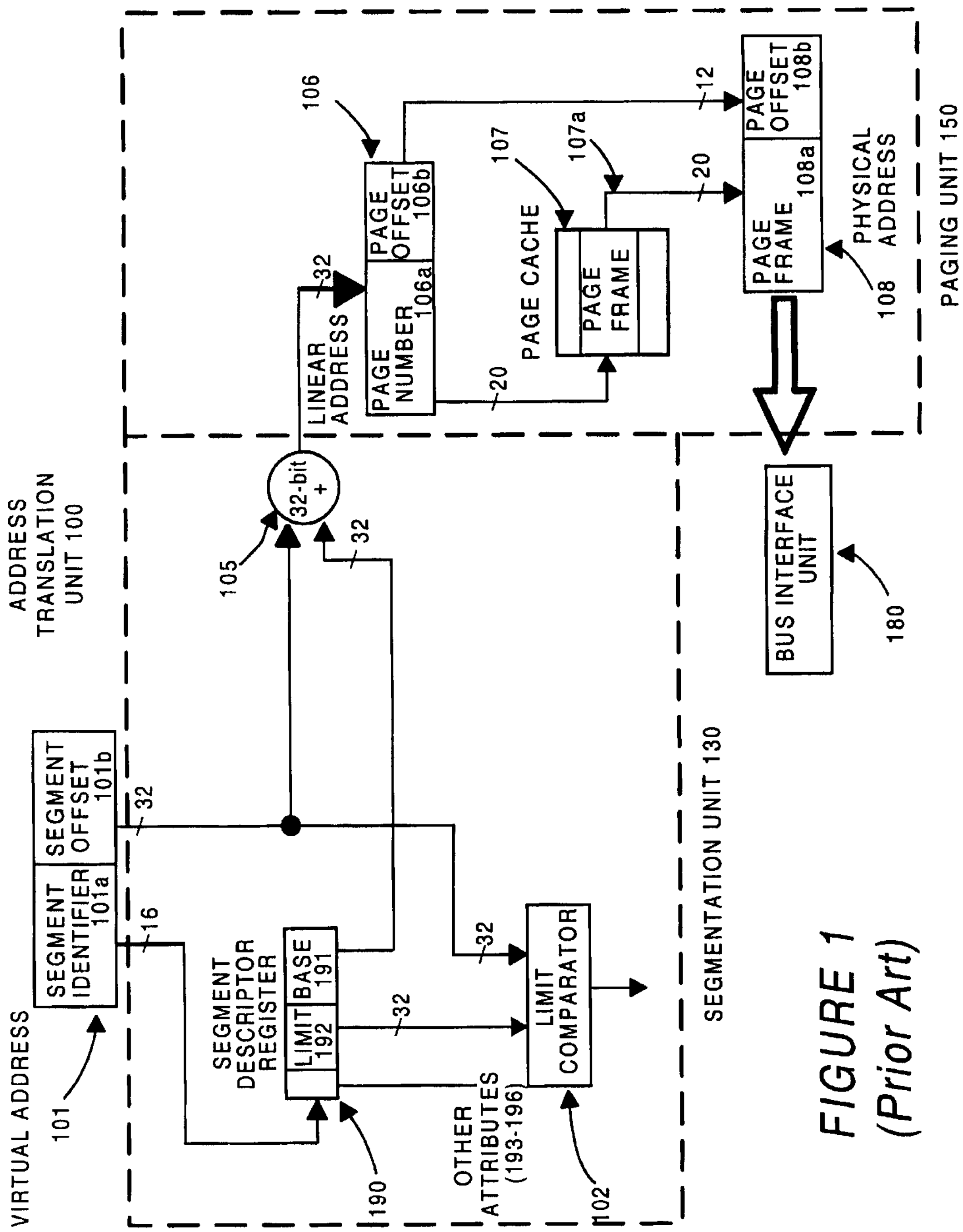
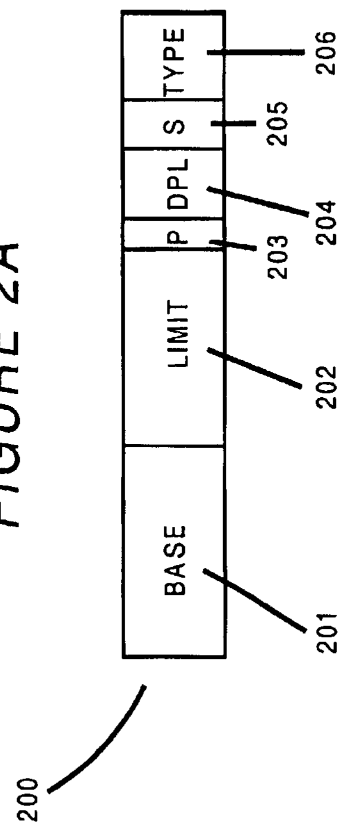


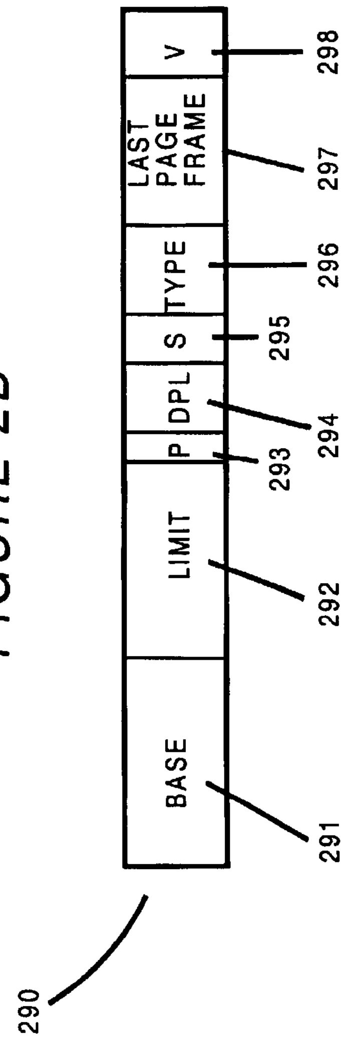
FIGURE 1
(Prior Art)

FIGURE 2A



PRIOR ART SEGMENT DESCRIPTOR REGISTER

FIGURE 2B



SEGMENT DESCRIPTOR MEMORY

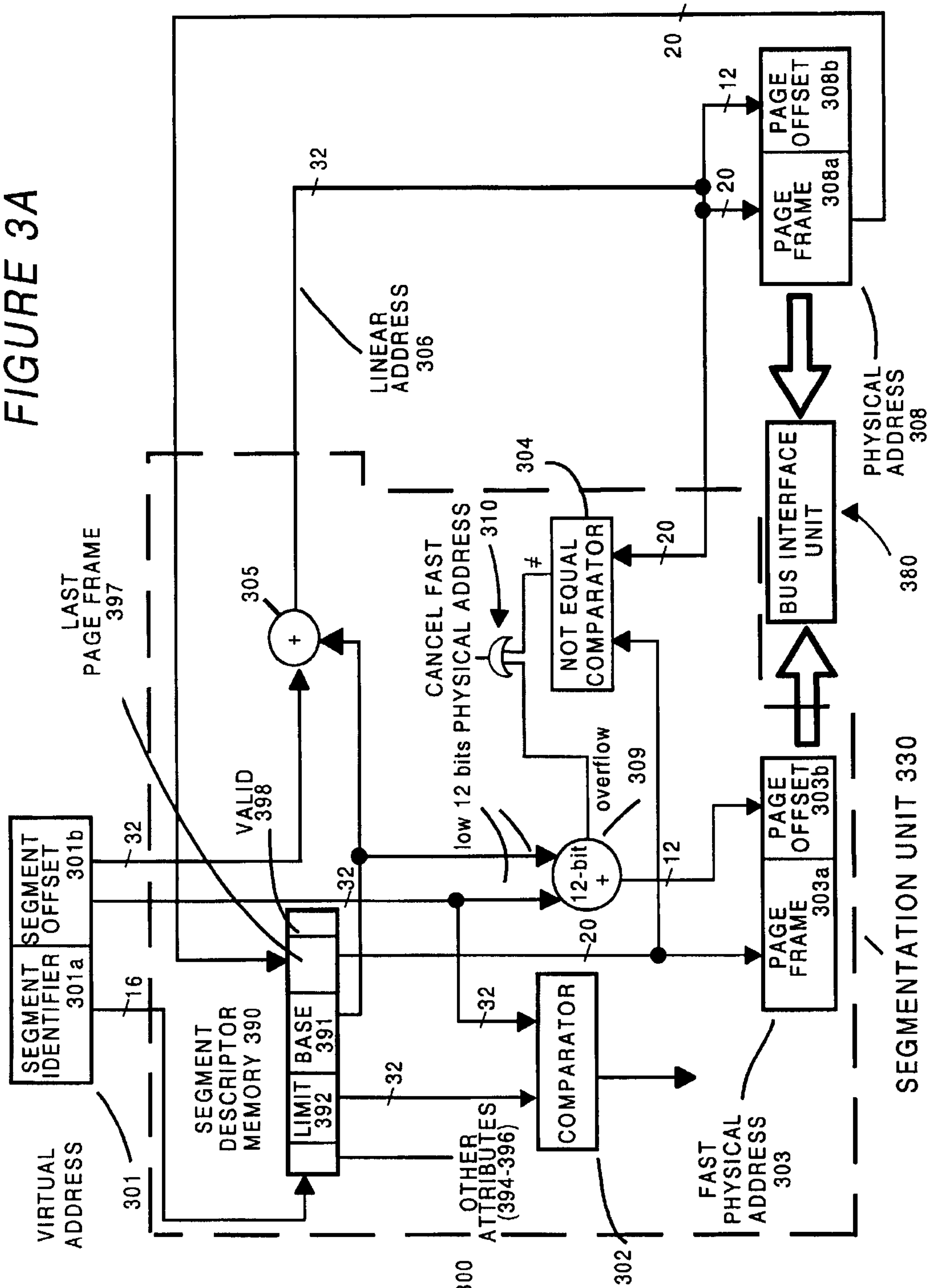

VIRTUAL ADDRESS
301
SEGMENT IDENTIFIER 301a
SEGMENT OFFSET 301b
16
32
LAST PAGE FRAME 397
VALID 398
SEGMENT DESCRIPTOR MEMORY 390
LIMIT 392
BASE 391
32
32
32
305
+
LINEAR ADDRESS 306
OTHER ATTRIBUTES (394-396)
COMPARATOR
302
300
low 12 bits
12-bit +
overflow
309
12
20
CANCEL FAST PHYSICAL ADDRESS 310
≠
NOT EQUAL COMPARATOR
304
20
20
12
PAGE FRAME 308a
PAGE OFFSET 308b
PHYSICAL ADDRESS 308
BUS INTERFACE UNIT
380
PAGE FRAME 303a
PAGE OFFSET 303b
FAST PHYSICAL ADDRESS 303
SEGMENTATION UNIT 330

FIGURE 3A

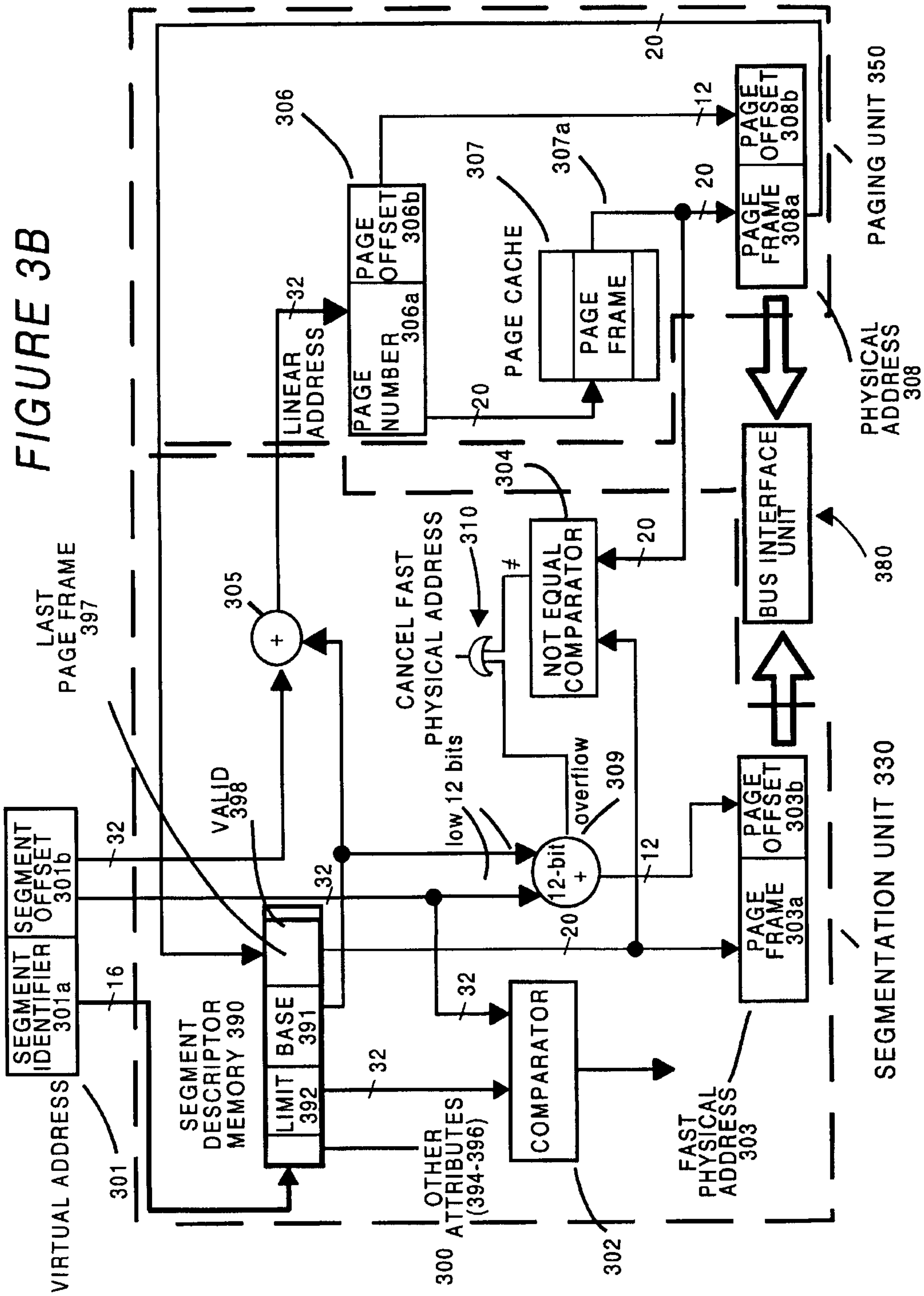
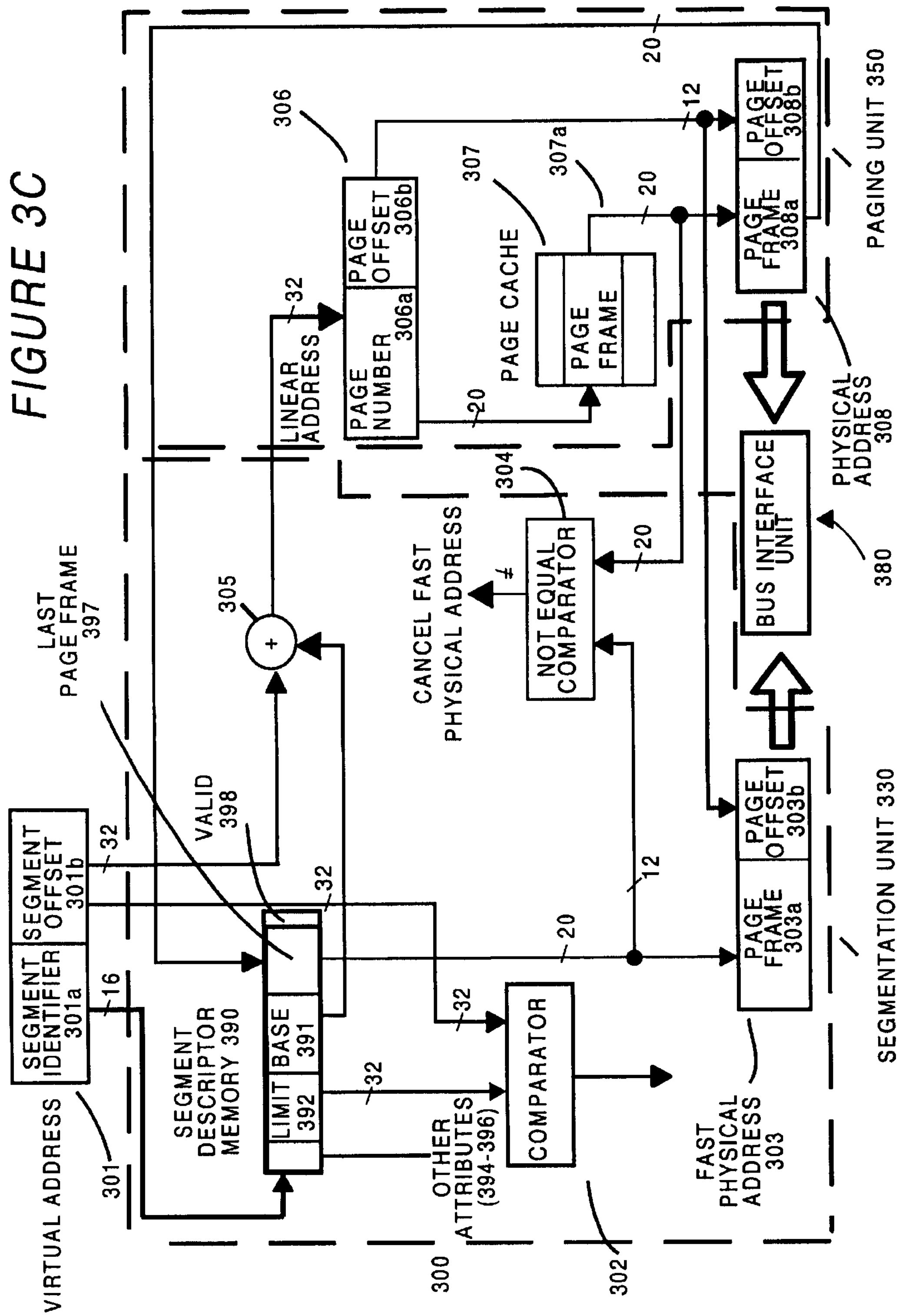

VIRTUAL ADDRESS
301
SEGMENT IDENTIFIER 301a
SEGMENT OFFSET 301b
16
32
LAST PAGE FRAME 397
VALID 398
SEGMENT DESCRIPTOR MEMORY 390
BASE 391
LIMIT 392
OTHER ATTRIBUTES (394-396)
300
COMPARATOR
302
305
+
LINEAR ADDRESS
PAGE NUMBER 306a
PAGE OFFSET 306b
306
PAGE CACHE
307
PAGE FRAME
307a
PAGE FRAME 308a
PAGE OFFSET 308b
PHYSICAL ADDRESS 308
PAGING UNIT 350
CANCEL FAST PHYSICAL ADDRESS
310
NOT EQUAL COMPARATOR
304
low 12 bits
overflow
309
12-bit +
BUS INTERFACE UNIT
380
PAGE FRAME 303a
PAGE OFFSET 303b
FAST PHYSICAL ADDRESS 303
SEGMENTATION UNIT 330
20
12

FIGURE 3B

FIGURE 3C


VIRTUAL ADDRESS
301
SEGMENT IDENTIFIER 301a
SEGMENT OFFSET 301b
16
32
SEGMENT DESCRIPTOR MEMORY 390
LIMIT 392
BASE 391
LAST PAGE FRAME 397
VALID 398
OTHER ATTRIBUTES (394-396)
COMPARATOR
302
300
305
LINEAR ADDRESS
PAGE NUMBER 306a
PAGE OFFSET 306b
306
PAGE CACHE
PAGE FRAME
307
307a
CANCEL FAST PHYSICAL ADDRESS
≠
NOT EQUAL COMPARATOR
304
20
12
BUS INTERFACE UNIT
380
PHYSICAL ADDRESS 308
PAGE FRAME 308a
PAGE OFFSET 308b
PAGING UNIT 350
PAGE FRAME 303a
PAGE OFFSET 303b
FAST PHYSICAL ADDRESS 303
SEGMENTATION UNIT 330

SPECULATIVE ADDRESS TRANSLATION FOR PROCESSOR USING SEGMENTATION AND OPTICAL PAGING

This application is a continuation of application Ser. No. 08/458,479 filed on Jun. 2, 1995, now U.S. Pat. No. 5,895, 503 and a continuation of application Ser. No. 09/905,410 now U.S. Pat. No. 5,960,466 filed Aug. 4, 1997. The present application is also a continuation of Ser. No. 08/905,356 filed Aug. 4, 1997, now U.S. Pat. No. 6,226,733.

FIELD OF THE INVENTION

The invention relates to the field of address translation for memory management m a computer system.

BACKGROUND OF THE INVENTION

Advanced computer hardware systems operate with complex computer software programs. Computer system designers typically separate the virtual address space, the address space used by programmers in their development of software, and the physical address space, the address space used by the computer system. This separation allows programmers to think in terms of their conceptual models, and to design computer software programs without reference to specific hardware implementations. During the actual execution of programs by the computer system, however, these separate addresses must be reconciled by translating software program virtual addresses into actual physical addresses that can be accessed in a computer memory subsystem.

There are many well known approaches for address translation in the memory management mechanism of a computer system. These approaches fall into basically two major categories: those which map the smaller virtual (sometimes called logical, symbolic or user) addresses onto larger physical or real memory addresses, and those which map larger virtual addresses onto smaller physical memory. Translation mechanisms of the former category are employed typically in minicomputers m which relatively small address fields (e.g.: 16 bit addresses) are mapped onto larger real memory. Translation mechanisms of the second category are used typically in microprocessors, workstations and mainframes. Within each of these categories, segmentation only, paging only, and a combination of segmentation and paging are well known for accomplishing the translation process.

The present invention is primarily directed to address translation mechanisms where larger virtual addresses are mapped onto smaller physical addresses, and further to systems where segmentation and optional paging is employed.

In a segmentation portion of an address translation system, the address space of a user program (or programs cooperatively operating as processes or tasks), is regarded as a collection of segments which have common high-level properties, such as code, data, stack, etc. The segmented address space is referenced by a 2-tuple, known as a virtual address, consisting of the following fields: <<s>, <d>>, where <s> refers to a segment number (also called identifier or locator), and <d> refers to a displacement or offset, such as a byte displacement or offset, within the segment identified by the segment number. The virtual address <17,421>, for example, refers to the 421st byte in segment **17**. The segmentation portion of the address translation mechanism, using information created by the operating system of the computer system, translates the virtual address into a linear address in a linear address space.

In a paging portion of an address translation system, a linear (or intermediate) address space consists of a group of pages. Each page is the same size (i.e. it contains the same number of addresses in the linear space). The linear address space is mapped onto a multiple of these pages, commonly, by considering the linear address space as the 2-tuple consisting of the following fields: <<page number >, <page offset >>. The page number (or page frame number) determines which linear page is referenced. The page offset is the offset or displacement, typically a byte offset, within the selected page.

In a paged system, the real (physical) memory of a computer is conceptually divided into a number of page frames, each page frame capable of holding a single page. Individual pages in the real memory are then located by the address translation mechanism by using one or more page tables created for, and maintained by, the operating system. These page tables are a mapping from a page number to a page frame. A specific page may or may not be present in the real memory at any point in time.

Address translation mechanisms which employ both segmentation and paging are well known in the art. There are two common subcategories within this area of virtual address translation schemes: address translation in which paging is an integral part of the segmentation mechanism; and, address translation in which paging is independent from segmentation.

In prior art address translation mechanisms where paging is an integral part of the segmentation mechanism, the page translation can proceed in parallel with the segment translation since segments must start at page boundaries and are fixed at an integer number of pages. The segment number typically identifies a specific page table and the segment offset identifies a page number (through the page table) and an offset within that page. While this mechanism has the advantage of speed (since the steps can proceed in parallel) it is not flexible (each segment must start at a fixed page boundary) and is not optimal from a space perspective (e.g. an integer number of pages must be used, even when the segment may only spill over to a fraction of another page).

In prior art address translation mechanisms where paging is independent from segmentation, page translation generally cannot proceed until an intermediate, or linear, address is first calculated by the segmentation mechanism. The resultant linear address is then mapped onto a specific page number and an offset within the page by the paging mechanism. The page number identifies a page frame through a page table, and the offset identifies the offset within that page. In such mechanisms, multiple segments can be allocated into a single page, a single segment can comprise multiple pages, or a combination of the above, since segments are allowed to start on any byte boundary, and have any byte length. Thus, in these systems, while there is flexibility in terms of the segment/page relationship, this flexibility comes at a cost of decreased address translation speed.

Certain prior art mechanisms where segmentation is independent from paging allow for optional paging. The segmentation step is always applied, but the paging step is either performed or not performed as selected by the operating system. These mechanisms typically allow for backward compatibility with systems in which segmentation was present, but paging was not included.

Typical of the prior art known to the Applicant in which paging is integral to segmentation is the Multics virtual memory, developed by Honeywell and described by the

book, "The Multics System", by Elliott Organick. Typical of the prior art known to the Applicant in which optional paging is independent from segmentation is that described in U.S. Pat. No. 5,321,836 assigned to the Intel Corporation, and that described in the Honeywell DPS-8 Assesmbly Instructions Manual. Furthermore, U.S. Pat. No. 4,084,225 assigned to the Sperry Rand Corporation contains a detailed discussion of general segmentation and paging techniques, and presents a detailed overview of the problems of virtual address translation.

Accordingly, a key limitation of the above prior art methods and implementations where segmentation is independent from paging is that the linear address must be fully calculated by the segmentation mechanism each time before the page translation can take place for each new virtual address. Only subsequent to the linear address calculation, can page translation take place. In high performance computer systems computer systems, this typically takes two full or more machine cycles and is performed on each memory reference. This additional overhead often can reduce the overall performance of the system significantly.

SUMMARY OF THE INVENTION

An object of the present invention, therefore, is to provide the speed performance advantages of integral segmentation and paging and, at the same time, provide the space compaction and compatibility advantages of separate segmentation and paging.

A further object of the present invention is to provide a virtual address translation mechanism which architecturally provides for accelerating references to main memory in a computer system which employs segmentation, or which employs both segmentation and optional paging.

Another object of the present invention is to provide additional caching of page information in a virtual address translation scheme.

An further object of the present invention is to provide a virtual address translation mechanism which reduces the number of references required to ensure memory access.

According to the present invention, a segmentation unit converts a vial address consisting of a segment identifier and a segment offset into a linear address. The segmentation unit includes a segment descriptor memory, which is selectable by the segment identifier. The entry pointed to by the segment identifier contains linear address information relating to the specific segment (i.e., linear address information describing the base of the segment referred to by the segment identifier, linear address information describing the limit of the segment referred to by the segment identifier, etc.) as well as physical address information pertaining to the segment—such as the page base of at least one of the pages represented by said segment.

In the above embodiment, unlike prior art systems, both segmentation and paging information are kept in the segmentation unit portion of the address translation system. The caching of this page information in the segmentation unit permits the address translation process to occur at much higher speed than in prior art systems, since the physical address information can be generated without having to perform a linear to physical address mapping in a separate paging unit.

The page base information stored in the segmentation unit is derived from the page frame known from the immediately prior in time address translation on a segment-by-segment basis. In order to complete the full physical address translation (i.e., a page frame number and page offset), the segmentation unit combines the page frame from the segment descriptor memory with the page offset field, and may store this result in a segmentation unit memory, which can be a memory table, or a register, or alternatively, it may generate the full physical address on demand.

This fast physical address generated by the segmentation unit based on the virtual address and prior page information can be used by a bus interface to access a physical location in the computer memory subsystem, even before the paging unit has completed its translation of the linear address into a page frame and page offset. Thus, fewer steps and references are required to create a memory access. Consequently, the address translation step occurs significantly faster. Since address translation occurs in a predominant number of instructions, overall system performance is improved.

The memory access is permitted to proceed to completion unless a comparison of the physical address information generated by the paging unit with the fast physical address generated by the segmentation unit shows that the page frame information of the segmentation unit is incorrect.

In alternative embodiments, the segmentation unit either generates the page offset by itself (by adding the lower portion of the segment offset and the segment base address) or receives it directly from the paging unit.

In further alternate embodiments, the incoming segment offset portion of the virtual address may be presented to the segmentation unit as components. The segmentation unit then combines these components in a typical base-plus-offset step using a conventional multiple input (typically 3-input) adder well known in the prior art.

As shown herein in the described invention, the segment descriptor memory may be a single register, a plurality of registers, a cache, or a combination of cache and register configurations.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a typical prior art virtual address translation mechanism using segmentation and independent paging.

FIG. 2A is a detailed diagram of a typical segment descriptor register of the prior art.

FIG. 2B is a detailed diagram of an embodiment of the present invention, including a portion of a segment descriptor memory used for storing physical address information;

FIG. 3A is a block diagram of an embodiment of the present invention employing segmentation and optional paging, and showing the overall structure and data paths used when paging is disabled during an address translation;

FIG. 3B is a block diagram of the embodiment of FIG. 3A showing the overall structure and data paths used when paging is enabled during an address translation;

FIG. 3C is a block diagram of another embodiment of the present invention, showing an alternative circuit for generating the fast physical address information.

DETAILED DESCRIPTION OF THE INVENTION

General Discussion of Paging & Segmentation

The present invention provides for improved virtual address translation in a computer system. The preferred embodiment employs the invention in a single-chip microprocessor system, however, it is well understood that such a virtual address translation system could be implemented in multiple die and chip configurations without departing from the spirit or claims of the present invention.

Before embarking on a specific discussion of the present invention, however, a brief explanation of the general principles of segmentation and paging follows in order to provide additional background information, and so that the teachings of the present invention may be understood in a proper context.

Referring to FIG. 1, a typical prior art virtual address translation mechanism **100** using both segmentation and, optionally, paging in a computer system is shown. As described in this figure, a data path within the microprocessor transmits a virtual address **101**, consisting of segment identifier **101***a* and a segment offset **101***b*, to segmentation unit **130**. Segments are defined by segment descriptor entries in at least one segment descriptor table or segment descriptor segment (not shown). Segment descriptor tables are created and managed by the operating system of the computer system, and are usually located in the memory subsystem. Segment descriptor entries are utilized in the CPU of the computer system by loading them into segment descriptor register **190** or a segment descriptor cache (not shown); the segment descriptor register/cache is usually internal to the CPU, and thus more quickly accessible by the translation unit.

In the paging unit **150**, pages are defined by a page table or multiple page tables (not shown), also created and managed by the operating system; again, these tables are also typically located in a memory subsystem. All or a portion of each page table can be loaded into a page cache **107** (within the CPU, sometimes called a translation look-aside buffer) to accelerate page references.

In operation, the segmentation unit **130** first translates a virtual address to a linear address and then (except in the case when optional paging is disabled) paging unit **150** translates the linear address into a real (or physical) memory address.

Typically (as in an x86 microprocessor) the segmentation unit translates a 48-bit virtual address **101** consisting of a 16-bit segment identifier (<s>) **101***a* and a 32-bit displacement within that segment (<d>) **101***b* to a 32-bit linear (intermediate) address **106**. The 16-bit segment identifier **101***a* uniquely identifies a specific segment; this identifier is used to access an entry in a segment descriptor table (not shown). In the prior art, this segment descriptor entry contains a base address of the segment **191**, the limit of the segment **192**, and other attribute information described further below. The segment descriptor entry is usually loaded into a segment descriptor register **190**.

Using adder **105**, the segmentation unit adds the segment base **191** of the segment to the 32-bit segment offset **101***b* in the virtual address to obtain a 32-bit linear address. The 32-bit segment offset **101***a* in the virtual address is also compared against the segment limit **192**, and the type of the access is checked against the segment attributes. A fault is generated and the addressing process is aborted if the 32-bit segment offset is outside the segment limit, or if the type of the access is not allowed by the segment attributes.

The resulting linear address **106** can be treated as an offset within a linear address space; and in the commonly implemented schemes of the prior art, these offsets are frequently byte offsets. When optional paging is disabled, the linear address **106** is exactly the real or physical memory address **108**. When optional paging is enabled, the linear address is treated as a 2- or 3-tuple depending on whether the paging unit **150** utilizes one or two level page tables.

In the 2-tuple case shown in FIG. **1**, which represents single level paging, the linear address, <<p>, <pd >> is divided into a page number field <p> **106***a*, and a page displacement (page offset) field within that page (<pd>) **106***b*. In the 3-tuple case (not shown) <<dp >, <p >, <pd>>, the linear address is divided into a page directory field (<dp>), a page number field <p> and a page displacement field <pd>. The page directory field indexes a page directory to locate a page table (not shown). The page number field indexes a page table to locate the page frame in real memory corresponding to the page number, and the page displacement field locates a byte within the selected page frame. Thus, paging unit **150** translates the 32-bit linear address **106** from the segmentation unit **130** to a 32-bit real (physical) address **108** using one or two level page tables using techniques which are well known in the art.

In all of the above prior art embodiments where segmentation is independent from paging, the segment descriptor table or tables of the virtual address translator are physically and logically separate from the page tables used to perform the described page translation. There is no paging information in the segment descriptor tables and, conversely, there is no segmentation information in the page tables.

This can be seen in FIG. **2**A. In this figure, a typical prior art segment descriptor entry **200**, is shown as it is typically used in a segment descriptor table or segment descriptor register associated with a segmentation unit. As can be seen there, segment descriptor **200** includes information on the segment base **201**, the segment limit **202**, whether the segment is present (P) **203** in memory, the descriptor privilege level (DPL) **204**, whether the segment belongs to a user or to the system (S) **205**, and the segment type **206** (code, data, stack, etc.)

For additional discussions pertaining to the prior art in segmentation, paging, segment descriptor tables, and page tables, the reader is directed to the references U.S. Pat. Nos. 5,408,626, 5,321,836, 4,084,225, which are expressly incorporated by reference herein.

Improve Segmentation Unit Using Paging Information

As shown in the immediate prior art, the paging and segmentation units (circuits) are completely separate and independent. Since the two units perform their translation sequentially, that is, the segment translation must precede the page translation to generate the linear address, high performance computer systems, such as those employing superscalar and superpipelined techniques, can suffer performance penalties. In some cases, it is even likely that the virtual address translation could fall into the systems' "critical path". The "critical path" is a well-known characteristic of a computer system and is typically considered as the longest (in terms of gate delay) path required to complete a primitive operation of the system.

Accordingly, if the virtual address translation is in the critical path, the delays associated with this translation could be significant in overall system performance. With the recognition of this consideration, the present invention includes page information in the segment translation process. The present invention recognizes the potential performance penalty of the prior art and alleviates it by storing paging information in the segmentation unit obtained from a paging unit in previous linear-to-real address translations.

As can be seen in FIG. **2**B, the present invention extends the segment descriptor entries of the prior art with a segment entry **290** having two additional fields: a LAST PAGE FRAME field **297** and a VALID field **298**. The LAST PAGE FRAME field **297** is used to hold the high-order 20 bits (i.e.: the page frame) of the real physical) memory address of the last physical address generated using the specified segment identifier. The VALID field **298** is a 1-bit field, and indicates whether or not the LAST PAGE FRAME field **297** is valid.

The remaining fields **291–296** perform the same function as comparable fields **201–206** respectively described above in connection with FIG. **2**A.

Segment descriptor tables (not shown) can be located in a memory subsystem, using any of the techniques well-known in the art. As is also known in the art, it is possible to speed up address translation within the segmentation unit by using a small cache, such as one or more registers, or associative memory. The present invention makes use of such a cache to store segment entries **290** shown above. Unlike the prior art, however, the segment entries **290** of the present invention each contain information describing recent physical address information for the specified segment. Accordingly, this information can be used by a circuit portion of the segmentation unit to generate a new physical address without going through the linear to physical mapping process typically associated with a paging unit.

While in some instances the physical address information may change between two time-sequential virtual addresses to the same segment (and thus, a complete translation is required by both the segmentation and paging units), in the majority of cases the page frame information will remain the same. Thus, the present invention affords a significant speed advantage over the prior art, because in the majority of cases a complete virtual-linear-physical address translation is not required before a memory access is generated.

Embodiment With Segmentation Optional Paging/Paging Disabled

Referring to FIG. **3**A, the advantage of using this new information in segment entry **290** in a segmentation unit or segmentation circuit is apparent from a review of the operation of an address translation. In this figure, a paging unit (or paging circuit) is disabled, as for example might occur only when a processor is used in a real mode of operation, rather than a protected mode of operation.

In a preferred embodiment, the present invention employs a segment descriptor memory comprising at least one, and preferably many, segment descriptor registers **390**, which are identical in every respect to the segment descriptor register described above in connection with FIG. **2**B. These segment descriptor registers are loaded from conventional segment descriptor tables or segment descriptor segments which are well known in the art. Each segment descriptor register **390** is loaded by the CPU before it can be used to reference physical memory. Segment descriptor register **390** can be loaded by the operating system or can be loaded by application programs. Certain instructions of the CPU are dedicated to loading segment descriptor registers, for example, the "LDS" instruction, "Load Pointer to DS Register". Loading by the operating system, or execution of instructions of this type, causes a base **391**, limit **392**, descriptor privilege level **394**, system/user indicator **395**, and type **396** to be loaded from segment tables or segment descriptor segments as in the prior art. The three remaining fields are present **393**, LAST PAGE FRAME **397** and VALID **398**. When a segment descriptor register **390** is loaded, present **393** is set to 1, indicating that the segment descriptor register **390** contents are present; the valid field **398** is set to 0, indicating that the last page frame number field **397** is not valid; and the LAST PAGE FRAME field **397** is not set, or may be set to 0.

After the loading of a segment descriptor register **390**, instructions of the CPU may make references to virtual memory; if a segment descriptor register is referenced before it is loaded, as indicated by present field **393** set to 0, a fault occurs and the reference to the segment descriptor register is aborted.

As explained above, the CPU makes references to virtual memory by specifying a 48-bit virtual address, consisting of a 16-bit segment identifier **301***a* and a 32-bit segment offset **301***b*. A data path within the CPU transmits virtual address **301** to the address translation mechanism **300**.

Segment descriptor memory **390** is indexed by segment number, so each entry in this memory containing data characteristics (i.e., base, access rights, limit) of a specific segment is selectable by the segment identifier from the virtual address. Assuming this is the first reference to physical memory specifying a newly loaded segment descriptor register, since the VALID bit **398** is set to false, a prior art virtual address translation takes place. This involves, among other things, as explained earlier, various validity checks (including checking attributes **394–396**, segment limit checking using comparator **302** and potentially others), and using adder **305** to add the segment descriptor's base address **391** to the segment offset **301***b* to calculate a linear address **306**.

While the implementation in the embodiment of FIG. **3**A shows the addition of the base address **391** to the segment offset **301***b* using adder **305** to generate the linear address **306**, it will be understood by those skilled in the art that this specific implementation of the virtual to linear address translation is not the only implementation of the present invention. In other implementations, the segment offset **301***b* might consist of one or more separate components. Different combinations of one or more of these components might be combined using well known techniques to form a linear address, such as one utilizing a three-input adder. The use of these components is discussed, for example, in U.S. Pat. No. 5,408,626, and that description is incorporated by reference herein.

As is well known, in this embodiment where paging is disabled, linear address **306** is also a physical address which can be used as the physical address **308**. Memory access control operations are not shown explicitly since they are only ancillary to the present invention, and are well described in the prior art. In general, however, a bus interface unit **380** is typically responsible for interactions with the real (physical) memory subsystem. The memory subsystem of a computer system employing the present invention preferably has timing and address and data bus transaction details which are desirably isolated from the CPU and address translation mechanism. The bus interface unit **380** is responsible for this isolation, and can be one of many conventional bus interface units of the prior art.

In the present invention, bus interface unit **380** receives the real memory address **308** from address translation mechanism **300** and coordinates with the real memory subsystem to provide data, in the case of a memory read request, or to store data, in the case of a memory write request. The real memory subsystem may comprise a hierarchy of real memory devices, such as a combination of data caches and dynamic RAM, and may have timing dependencies and characteristics which are isolated from the CPU and address translation mechanism **300** of the computer system by the bus interface unit **380**.

Simultaneous with the first memory reference using the calculated physical address **308**, the LAST PAGE FRAME field of the selected segment descriptor register **390** is loaded with the high-order 20 bits of the physical address, i.e.: the physical page frame, and the VALID bit is set to indicate a valid state. This paging information will now be used in a next virtual address translation.

Accordingly, when a next, new virtual address **301** is to be translated, the entry selected from segment descriptor

memory **390** will likely contain the correct physical frame page number (in the LAST PAGE FRAME field **397**). Thus, in most cases, the base physical address in memory for the next, new referenced virtual address will also be known from a previous translation.

The first step of the virtual address translation, therefore, is to determine if a FAST PHYSICAL ADDRESS **303** can be used to begin a fast physical memory reference. Adder **309**, a 12-bit adder, adds the low-order 12 bits of the segment offset **301***b* of virtual address **301** to the low-order 12-bits of base **391** of the segment entry in segment descriptor register **390** referenced by the segment identifier **301***a*. This addition results in a page offset **303***b*. In parallel with adder **309**, 32-bit adder **305** begins a full 32-bit add of segment base **391** and segment offset **301***b*, to begin producing the linear address; however, this full 32-bit add will obviously require more time. In the preferred embodiment, adder **309** is a separate 12-bit adder; however, it should be noted that adder **309** also could be implemented as the low order 12-bits of 32-bit adder **305**.

Simultaneous with the beginning of these two operations, VALID bit **398** is inspected. If VALID bit **398** is set to 1, as soon as 12-bit adder **309** has completed, 20-bit LAST PAGE FRAME **397** is concatenated with the result of adder **309** to produce FAST PHYSICAL ADDRESS **303**, consisting of a page frame number **303***a*, and page offset **303***b*. FAST PHYSICAL ADDRESS **303** then can be used to tentatively begin a reference to the physical memory. It should be understood that the FAST PHYSICAL ADDRESS **303** transmitted to bus interface unit **380** could also be stored in a register or other suitable memory storage within the CPU.

In parallel with the fast memory reference, limit field **392** is compared to the segment offset **301***b* of the virtual address by comparator **302**. If the offset in the virtual address is greater than the limit, a limit fault is generated, and virtual address translation is aborted.

Also in parallel with the fast memory reference, adder **305** completes the addition of base **391** to the segment offset field **301***b* of virtual address to produce linear address (in this case physical address also) **306**. When this calculation is completed, the page frame number **308***a* of physical address **308** is compared to LAST PAGE FRAME **397** by Not Equal Comparator **304**. If page frame **308***a* is unequal to the LAST PAGE FRAME **397**, or if 12-bit Adder **309** overflowed (as indicated by a logic "1" at OR gate **310**), the fast memory reference is canceled, and the linear address **306**, which is equal to the physical address **308**, is used to begin a normal memory reference. If page frame **308***a* is equal to LAST PAGE FRAME **397**, and 12-bit Adder **309** did not overflow (the combination indicated by a logic "0" at the output of OR gate **310**), the fast memory reference is allowed to fully proceed to completion.

After any fast memory reference which is cancelled by the CANCEL FAST PHYSICAL ADDRESS signal output of OR gate **310**, page frame **308***a* is loaded into the LAST PAGE FRAME **397** in the segment descriptor memory **390** for subsequent memory references.

Depending on the particular design desired, it should also be noted that writes to the memory, or reads which cause faults using FAST PHYSICAL ADDRESS **303** may be pended since the FAST PHYSICAL ADDRESS **303** may prove to be invalid.

Accordingly, it can be seen that the parallel physical address calculation undertaken by the improved segmentation unit of the present invention generates a faster physical memory access than possible with prior art systems.

Embodiment With Segmentation & Paging/Paging Enabled

The present invention can also be used with address translation units using paging enabled, as can be seen in the embodiments of FIGS. **3B** and **3C**.

In the embodiment of FIG. **3B**, the same segmentation unit structure **300** as that shown in FIG. **3A** is used, and the operation of segmentation unit **300** is identical to that already explained above. As before, segment descriptor memory (registers) **390** are loaded from conventional segment descriptor tables or segment descriptor segments, using one or more of the procedures described above. First, the base **391** limit **392** descriptor privilege level **394**, system/user indicator **395**, and type **396** are loaded from segment tables or segment descriptor segments as explained earlier. When segment descriptor register **390** is loaded, present **393** is set to 1, indicating that the segment descriptor register **390** contents are present; the valid field **398** is set to 0, indicating that the last page frame number field **397** is not valid; and the LAST PAGE FRAME field **397** is not set, or may be set to 0.

As explained above, after the loading of a segment descriptor register **390**, instructions of the CPU may make references to virtual memory; if a segment descriptor register is referenced before it is loaded, as indicated by present field **393** set to 0, a fault occurs and the reference to the segment descriptor register is aborted.

As further explained above, the 48 bit virtual address **301** (consisting of a 16 bit segment identifier **301***a* and a 32 bit segment offset **301***b*) is transmitted by a data path to segmentation unit **300**, and an index into segment descriptor memory **390** is performed to locate the specific segment descriptor for the segment pointed to by segment identifier **301***a*. Assuming this is the first reference to physical memory specifying a newly loaded segment descriptor register, since the VALID bit is set to false, a prior art virtual address translation takes place. This involves, among other things, as explained earlier, various validity checks (including checking attributes **394–396**, segment limit checking using comparator **302** and potentially others), and using adder **305** to add the segment descriptor's base address **391** to the segment offset **301***b* to calculate a linear address **306**.

As is well known, in this configuration where paging is enabled, linear address **306** must undergo a further translation by paging unit **350** to obtain the physical address **308** in the memory subsystem. In the preferred embodiment of the invention, looking first at FIG. **3B**, the output of adder **305** will be a 32-bit linear address, corresponding to a 20-bit page number **306***a* and a 12-bit page offset **306***b*. Typically, the page number **306***a* is then indexed into a page descriptor table (not shown) to locate the appropriate page frame base physical address in memory. These page descriptor tables are set up by the operating system of the CPU using methods and structures well known in the art, and they contain, among other things, the base physical address of each page frame, access attributes, etc.

However, in most systems, including the present invention, a page cache **307** is used in order to hold the physical base addresses of the most recently used page frames. This cache can take the form of a table, associative cache, or other suitable high speed structure well known in the art, Thus, page number **306***a* is used to access page data (including physical base addresses for page frames) in an entry in page cache **307**.

If page cache **307** hits, two things happen: first, a 20-bit PAGE FRAME **307***a* (the page frame in physical memory) replaces the high-order 20 bits (page number **306***a*) of the linear address **306**, and, when concatenated with the page

offset **306***b* results in a real (physical) address **308**, which is used to perform a memory access through bus interface unit **380** along the lines explained above. Second, newly generated page frame **308***a* is also stored in segment descriptor memory **390** in the selected LAST PAGE FRAME field **397** to be used for a fast access in the next address translation. When LAST PAGE FRAME field **397** is stored, selected VALID bit **398** is set to 1 to indicate that LAST PAGE FRAME **397** is valid for use.

In the event of a page cache miss, the appropriate page frame number **308***a* is located (using standard, well-known techniques) to generate physical address **308**, and is also loaded into segment descriptor memory **390** in the selected LAST PAGE FRAME field **397**. The selected VALID bit **398** is also set to indicate a valid state. Thus, there is paging information in the segmentation unit that will now be used in the next virtual address translation.

When a next, new virtual address **301** is to be translated, the segment identifier **301***a* will likely be the same as that of a previously translated virtual address, and the entry selected from segment descriptor memory **390** will also likely contain the correct physical frame (in LAST PAGE FRAME field **397**) from the previous translation. As with the above embodiment, one or more registers, or a cache may be used for the segment descriptor memory **390**.

The first step then determines if a FAST PHYSICAL ADDRESS **303** can be used to begin a fast physical memory reference. Adder **309**, a 12-bit adder, adds the low-order 12 bits of the segment offset **301***b* of virtual address **301** to the low-order 12-bits of base **391** of the segment entry in segment descriptor register **390** referenced by the segment identifier **301***a*. This addition results in a page offset **303***b*. In parallel with adder **309**, 32-bit adder **305** begins a full 32-bit add of segment base **301** and segment offset **301***b*, to begin producing the linear address; however, this full 32-bit add will obviously require more time. In the preferred embodiment, adder **309** is a separate 12-bit adder; however, it should be noted that adder **309** also could be implemented as the low order 12-bits of 32-bit adder **305**.

Simultaneous with these beginning of these two operations, VALID bit **398** is inspected. If VALID bit **398** is set to 1, as soon as 12-bit adder **309** has completed, 20-bit LAST PAGE FRAME **397** is concatenated with the result of adder **309** to produce FAST PHYSICAL ADDRESS **303**, consisting of a page frame number **303***a*, and page offset **303***b*. FAST PHYSICAL ADDRESS **303** then can be used to tentatively begin a reference to the physical memory. Again, it should be understood that the FAST PHYSICAL ADDRESS **303** transmitted to bus interface unit **380** could also be stored in a register or other suitable memory storage within the CPU.

As before, limit field **302** is compared to the segment offset **301***b* of the virtual address by comparator **302**. If the offset in the virtual address is greater than the limit, a limit fault is generated, and virtual address translation is aborted.

This new virtual address is also translated by paging unit **350** in the same manner as was done for the previous virtual address. If page cache **307** hits based on the page number **306***a*, two things happen: first, a 20-bit PAGE FRAME **307***a* (the page frame in physical memory) replaces the high-order 20 bits (page number **306***a*) of the linear address **306**, and, when concatenated with the page offset **306***b* results in a physical address **308**. This real address may or may not be used, depending on the result of the following: in parallel with the aforementioned concatenation, the PAGE FRAME **307***a*, is compared to LAST PAGE FRAME **397** from the segment descriptor memory **390** by Not Equal Comparator **304**. The result of Not Equal Comparator (that is, the Not Equal condition) is logically ORed with the overflow of 12-bit adder **309** by OR gate **310**. If the output of OR gate **310** is true (i.e. CANCEL FAST PHYSICAL ADDRESS is equal to binary one), or if PAGE CACHE **307** indicates a miss condition, the fast memory reference previously begun is canceled, since the real memory reference started is an invalid reference. Otherwise, the fast memory reference started is allowed to fully proceed to completion, since it is a valid real memory reference.

If CANCEL FAST PHYSICAL ADDRESS is logical true, it can be true for one of two, or both reasons. In the case that Or gate **310** is true, but page cache **307** indicates a hit condition, physical address **308** is instead used to start a normal memory reference. This situation is indicative of a situation where LAST PAGE FRAME **397** is different from the page frame **308***a* of the current reference.

In the case that page cache **307** did not indicate a hit, a page table reference through the page descriptor table is required and virtual address translation proceeds as in the prior art. The page frame **308***a* information is again stored in the LAST PAGE FRAME field **397** in the segment descriptor memory **390** for the next translation.

Also, after any fast memory reference which is canceled by the CANCEL FAST PHYSICAL ADDRESS signal output of OR gate **310**, page frame number **308***a* is loaded into the LAST PAGE FRAME **397** in the segment descriptor memory **390** for subsequent memory references.

Depending on the particular design desired, it should also be noted that in this embodiment also, writes to the memory, or reads which cause faults using FAST PHYSICAL ADDRESS **303** may be pended since the FAST PHYSICAL ADDRESS **303** may prove to be invalid.

The alternative embodiment shown in FIG. 3C is identical in structure and operation to the embodiment of FIG. 3B, with the exception that the 12-bit Adder **309**, is not employed. In this embodiment, the segmentation unit **330** does not create the lower portion (page offset **303***a*) of the fast physical address in this manner. Instead, the page offset **306***a* resulting from 32-bit adder **305** is used.

It can be seen that the present invention has particular relevance to computers using sequential type of segmentation and paging translation, such as the X86 family of processors produced by the Intel Corporation (including the Intel 80386, Intel 80486 and the Intel Pentium Processor), other X86 processors manufactured by the NexGen Corporation, Advanced Micro Devices, Texas Instruments, International Business Machines, Cyrix Corporation, and certain prior art computers made by Honeywell. These processors are provided by way of example, only, and it will be understood by those skilled in the art that the present invention has special applicability to any computer system where software executing on the processors is characterized by dynamic execution of instructions in programs in such a way that the virtual addresses are generally logically and physically located near previous virtual addresses.

The present invention recognizes this characteristic, employing acceleration techniques for translating virtual to real addresses. In particular, the present invention utilizes any of the commonly known storage structures (specific examples include high speed registers and/or caches) to store previous address translation information, and to make this previous address translation information available to the system whenever the next subsequent reference relies on the same information. In this way, the system can utilize the previously stored information from the high speed storage to begin real memory references, rather than be forced to

execute a more time consuming translation of this same information, as was typically done in the prior art.

As will be apparent to those skilled in the art, there are other specific circuits and structures beyond and/or in addition to those explicitly described herein which will serve to implement the translation mechanism of the present invention. Finally, although the above description enables the specific embodiment described herein, these specifics are not intended to restrict the invention, which should only be limited as defined by the following claims.

I claim:

1. A system for performing address translations in a processor employing both segmentation and optional independent paging the system comprising:

a page cache providing an actual physical page frame address from a virtual address in a time period T, the page frame cache accessed by using a page field of a fully calculated linear address; and

a speculative physical page frame address generator providing a speculative physical page address related to said virtual address in a time <T;

wherein the respective page frames are combined with offset portions to produce physical memory addresses.

2. The system of claim **1**, wherein the speculative physical page frame address can be used for generating a memory access faster than a memory access based on said actual physical page frame address.

3. The system of claim **2** wherein the memory access is to a cache memory.

4. The system of claim **2** including a cancellation circuit for canceling the memory access if the speculative physical page frame address and actual physical page frame address are different.

5. The system of claim **1**, wherein the speculative physical page frame address generator comprises a second page frame cache.

6. The system of claim **1** wherein the speculative physical page frame address generator comprises a page frame address cache accessed during calculation of said fully calculated linear address.

7. A circuit for performing memory accesses in a microprocessor system that uses a virtual address having a segment identifier and a segment offset, the circuit comprising:

a) a linear address generator adapted to calculate a calculated linear address based on processing the entire virtual address; and

b) a physical address generator, coupled to the linear address generator, adapted to generate a calculated physical address based on processing all of said calculated linear address, said physical address generator including a first memory for caching said calculated physical address; and

c) a second memory coupled to the linear address generator storing physical address information usable to generate a tentative physical address;

wherein said tentative physical address is used to initiate a tentative memory access that is completed unless said tentative physical address is different from said calculated physical address, in which case said calculated physical address is instead used for a calculated memory access.

8. The circuit of claim **7**, wherein said tentative physical address and said calculated physical address are generated in parallel, and said tentative physical address is completed before said calculated physical address can be completed by said physical address generator.

9. The circuit of claim **8**, wherein said tentative physical address is generated based on processing a portion of said calculated linear address.

10. The circuit of claim **9**, wherein said calculated linear address is a 32 bit linear address, and said tentative physical address is based on a lower portion of said 32 bit linear address.

11. The circuit of claim **7**, wherein said physical address information in said second memory is derived from a translation of a prior virtual address.

12. The circuit of claim **7** wherein said first memory is a page cache located in a paging unit of the microprocessor.

13. The circuit of claim **7**, wherein said second memory includes segment descriptor information.

14. The circuit of claim **7**, wherein the tentative memory access and calculated memory access are to a cache memory.

15. A computer system using segmentation and optional independent paging for performing address translations comprising:

an address translation memory capable of storing a portion of a physical address corresponding to a stored page frame;

a virtual to linear address converter circuit for generating a calculated linear address; and

a linear to physical address converter circuit for receiving and generating a calculated physical address based on the calculated linear address, the calculated physical address including a first page frame and a first page offset; and

a fast physical address circuit generating a fast physical address comprised of the stored page frame combined with a page offset portion derived from the virtual address;

wherein the fast physical address is generated prior to the generation of said calculated physical address.

16. The system of claim **15**, wherein the fast physical address can be used to initiate a fast memory access sooner than a memory access resulting from said first physical address.

17. The system of claim **16**, including a cancellation circuit for canceling the fast memory access if the fast physical address and first physical address are different.

18. The circuit of claim **17**, wherein the fast physical address is generated during the generation of the first linear address.

19. The system of claim **18**, wherein the stored page frame is generated in a prior address translation based on a prior virtual address.

20. A method of performing memory references in a processor that employs both segmentation and optional independent paging during an address translation, said system comprising;

performing an actual address translation from a virtual address by first calculating a linear address based on both a segment identifier and an offset associated with the virtual address, and then generating an actual physical address based on the calculated linear address; and

performing a speculative address translation from the virtual address using portions of the linear address and actual physical address information from a prior virtual address translation to produce a speculative physical address;

performing a memory reference using the speculative physical address; validating that the memory reference is valid.

21. The method of claim **20** wherein the validating step comprises comparing the page frame portions of the actual physical address and the speculative physical address.

22. The method of claim **21**, further including a step of canceling the memory reference if the page frame portions

of the actual physical address and the speculative physical address are different.

23. A method of performing memory accesses in a microprocessor system using a virtual address having a segment identifier and a segment offset, the method comprising the steps of:

(a) generating a calculated linear address based on processing said entire virtual address;

(b) using a first cache containing physical address information to generate a calculated physical address based on said calculated linear address;

(c) using a second cache containing physical address information to generate a tentative physical address in parallel with step (a) and before step (b) is completed, said tentative physical address being based in part on a portion of said calculated linear address;

(d) using said tentative physical address to initiate a tentative memory access to a cache;

(e) completing said tentative memory access to said cache when said tentative physical address and said calculated physical address are the same;

(f) aborting said tentative memory access and performing a second memory access based on said calculated physical address when said tentative physical address and said calculated physical address are different.

24. The method of claim **23**, wherein said first cache is a page cache located in a paging unit of the microprocessor.

25. The method of claim **23**, wherein said second cache includes segment descriptor information.

26. The method of claim **23**, wherein said calculated linear address is a 32 bit linear address, and said tentative physical address is based on a lower portion of said 32 bit linear address.

* * * * *

EXHIBIT E

US006813699B1

(12) United States Patent
Belgard

(10) Patent No.: US 6,813,699 B1
(45) Date of Patent: Nov. 2, 2004

(54) **SPECULATIVE ADDRESS TRANSLATION FOR PROCESSOR USING SEGMENTATION AND OPTIONAL PAGING**

(75) Inventor: **Richard Belgard**, Saratoga, CA (US)

(73) Assignee: **Transmeta Corporation**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **10/166,432**

(22) Filed: **Jun. 10, 2002**

Related U.S. Application Data

(63) Continuation of application No. 09/757,439, filed on Jan. 10, 2001, now Pat. No. 6,430,668, which is a continuation of application No. 08/905,356, filed on Aug. 4, 1997, now Pat. No. 6,226,733, which is a continuation of application No. 08/458,479, filed on Jun. 2, 1995, now Pat. No. 5,895,503.

(51) **Int. Cl.**[7] **G06F 12/02**
(52) **U.S. Cl.** **711/213**; 711/203; 711/204; 711/205; 711/206; 711/207; 711/209; 711/220
(58) **Field of Search** 711/203, 204, 711/205, 206, 207, 209, 213, 220

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,084,225 A		4/1978	Anderson et al.	
4,400,774 A		8/1983	Toy	
5,165,028 A		11/1992	Zulian	
5,321,836 A	*	6/1994	Crawford et al.	711/206
5,335,333 A		8/1994	Hinton et al.	
5,423,014 A		6/1995	Hinton et al.	
5,895,503 A	*	4/1999	Belgard	711/202
5,900,022 A	*	5/1999	Kranich	711/205
5,960,466 A	*	9/1999	Belgard	711/213
5,961,580 A	*	10/1999	Mahalingaiah	708/670
6,226,733 B1	*	5/2001	Belgard	711/213
6,430,668 B2	*	8/2002	Belgard	711/202

FOREIGN PATENT DOCUMENTS

EP 0668565 A1 8/1995

OTHER PUBLICATIONS

T.M.Austin, D.N. Pnevmatikos, G.S. Sohi, "Streamlining Data Cache Access with Fast Address Calculation," In 22nd International Symposium on Computer Architecture, 1995, 12 pages, Jun. 22, 1995.

J. Bradley Chen, Anita Borg, and Norman P. Jouppi, "A Simulation Based Study of TLB Performance," In Proc. 19th ISCA. ACM, 1992, 32 pages.

"U5S 486 Green CPU," United Microelectronics Corporation, 1994–95, pp. 3–1 to 3–26.

"Intel Microprocessors," vol. 1, Intel Corporation, 1993, pp. 2–229 to 2–287.

"Computer Architecture A Quantitative Approach," Hennessey and Patterson, pp. 432–497 (1990).

Hua, A. Hunt, L. Liu, J–K Peir, D. Pruett, and J. Temple, "Early Resolution of Address Translation in Cache Design," Proc. of Int'l Conf. on Computer Designs, Oct. 1990, pp. 408–412.

"DPS–8 Assembly Instructions," Honeywell Corporation, Apr., 1980, Chapters 3 and 5.

"The Multics System," Elliot Organick, 1972, pp. 6–7, 38–51.

* cited by examiner

Primary Examiner—T Nguyen
(74) *Attorney, Agent, or Firm*—J. Nicholas Gross

(57) **ABSTRACT**

An improved address translation method and mechanism for memory management in a computer system is disclosed. A segmentation mechanism employing segment registers maps virtual addresses into a linear address space. A paging mechanism optionally maps linear addresses into physical or real addresses. Independent protection of address spaces is provided at each level. Information about the state of real memory pages is kept in segment registers or a segment register cache potentially enabling real memory access to occur simultaneously with address calculation, thereby increasing performance of the computer system.

15 Claims, 5 Drawing Sheets

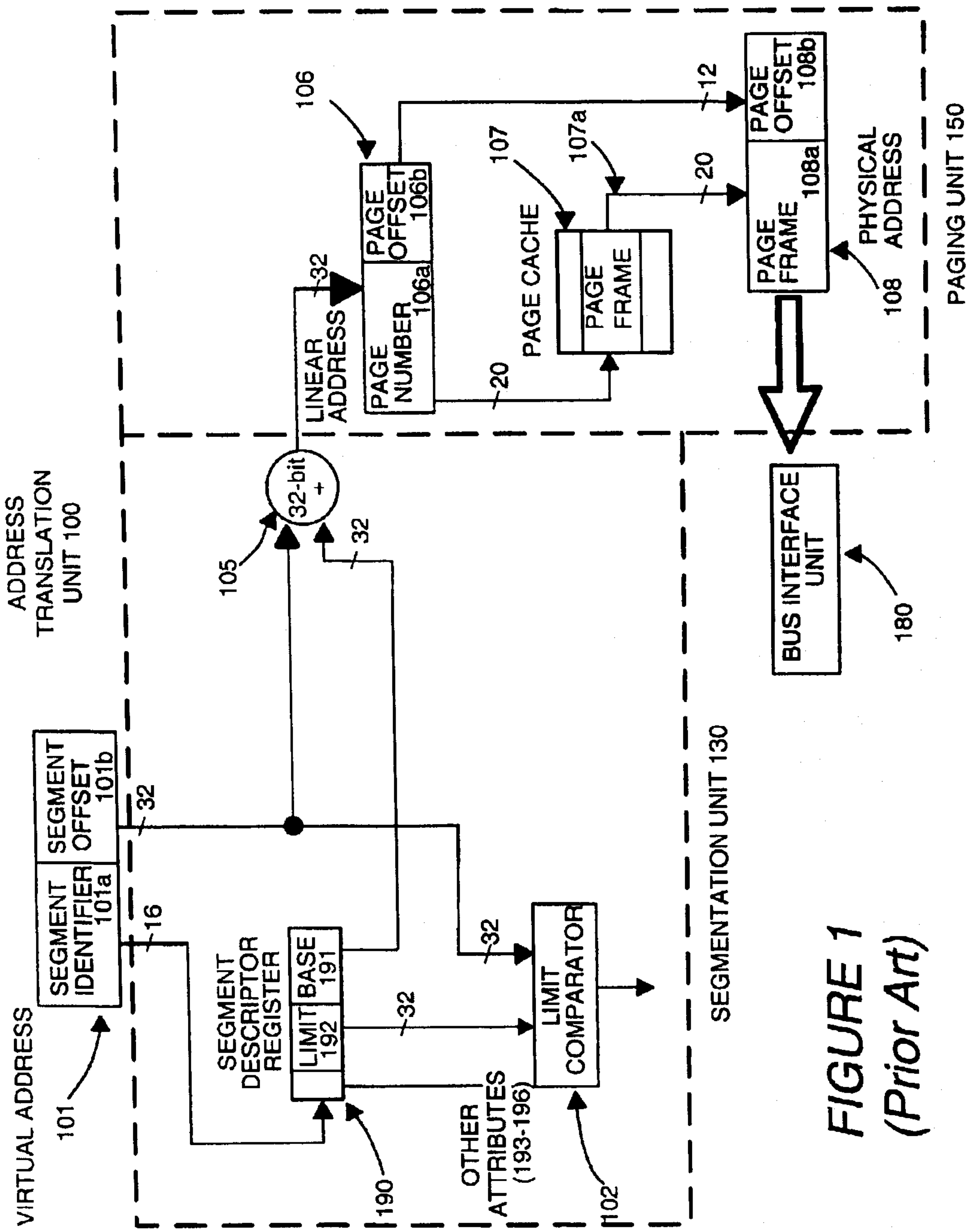
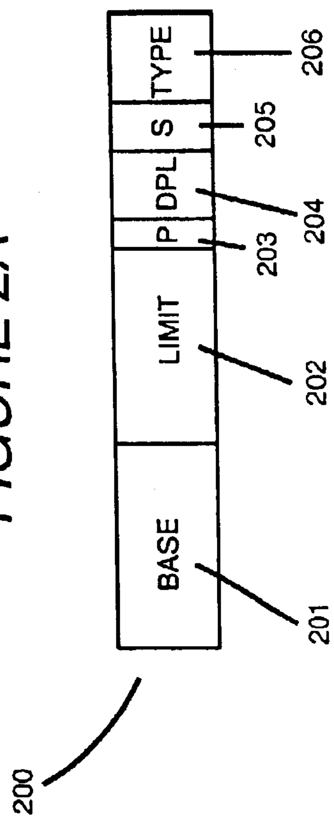


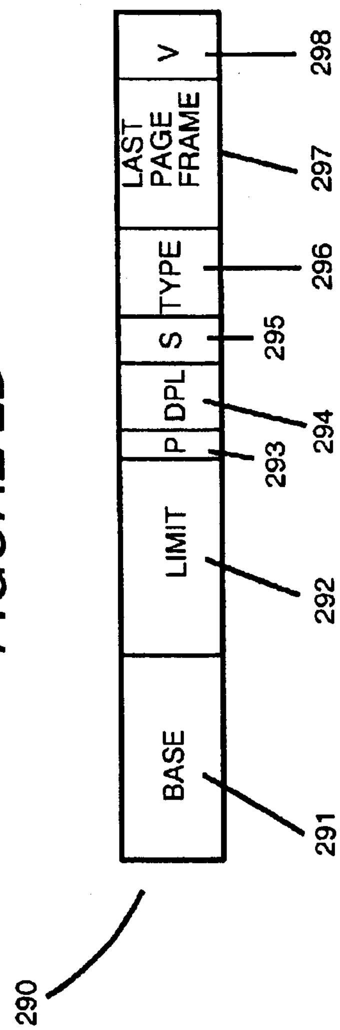
FIGURE 1
(Prior Art)

FIGURE 2A



PRIOR ART SEGMENT DESCRIPTOR REGISTER

FIGURE 2B



SEGMENT DESCRIPTOR MEMORY

FIGURE 3A

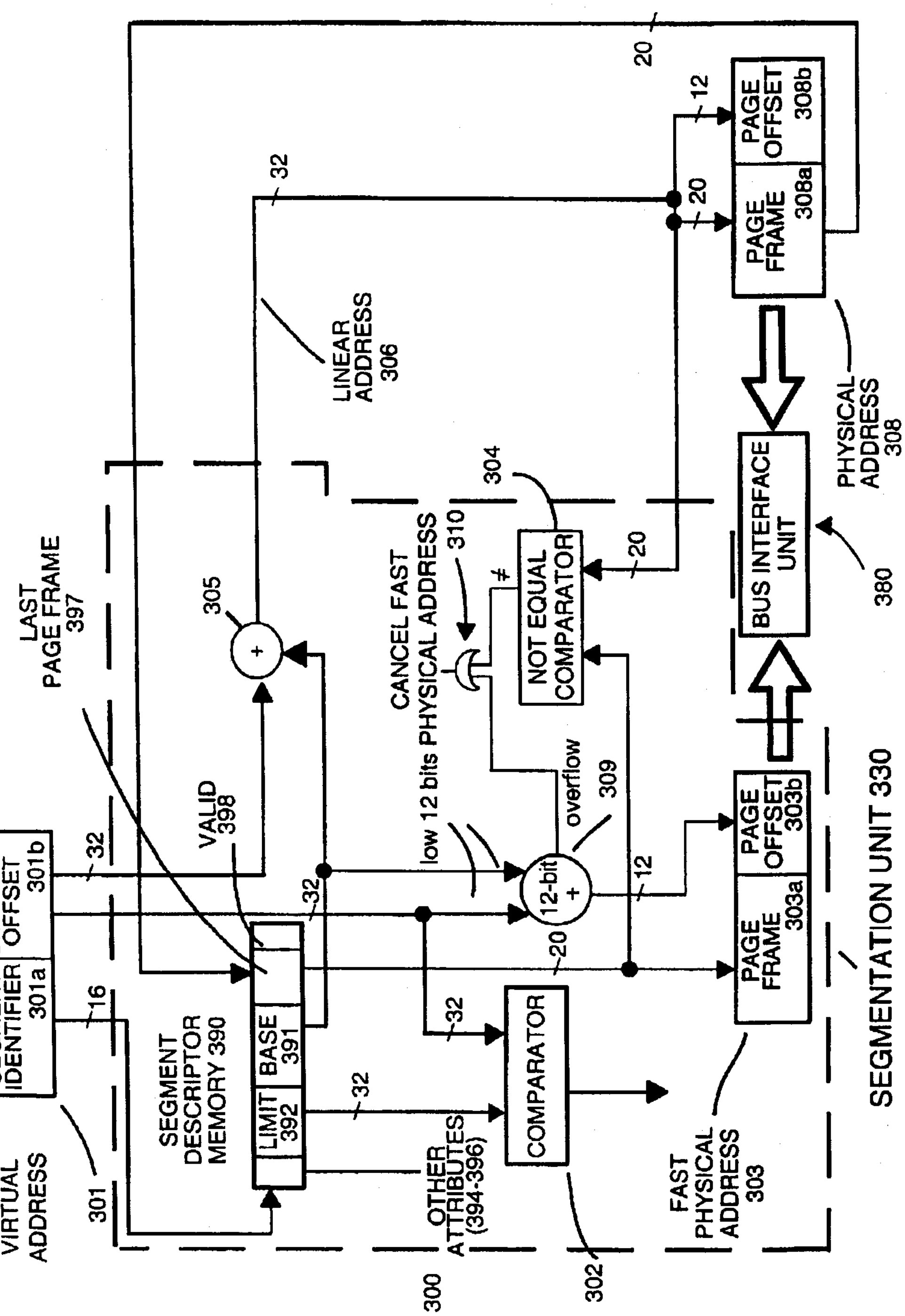

VIRTUAL ADDRESS
301
SEGMENT IDENTIFIER 301a
SEGMENT OFFSET 301b
16
32
LAST PAGE FRAME 397
VALID 398
SEGMENT DESCRIPTOR MEMORY 390
LIMIT 392
BASE 391
OTHER ATTRIBUTES (394-396)
COMPARATOR
302
300
305
+
LINEAR ADDRESS 306
low 12 bits
12-bit +
overflow
309
CANCEL FAST PHYSICAL ADDRESS
310
#
NOT EQUAL COMPARATOR
304
20
12
FAST PHYSICAL ADDRESS 303
PAGE FRAME 303a
PAGE OFFSET 303b
SEGMENTATION UNIT 330
BUS INTERFACE UNIT
380
PHYSICAL ADDRESS 308
PAGE FRAME 308a
PAGE OFFSET 308b

FIGURE 3B

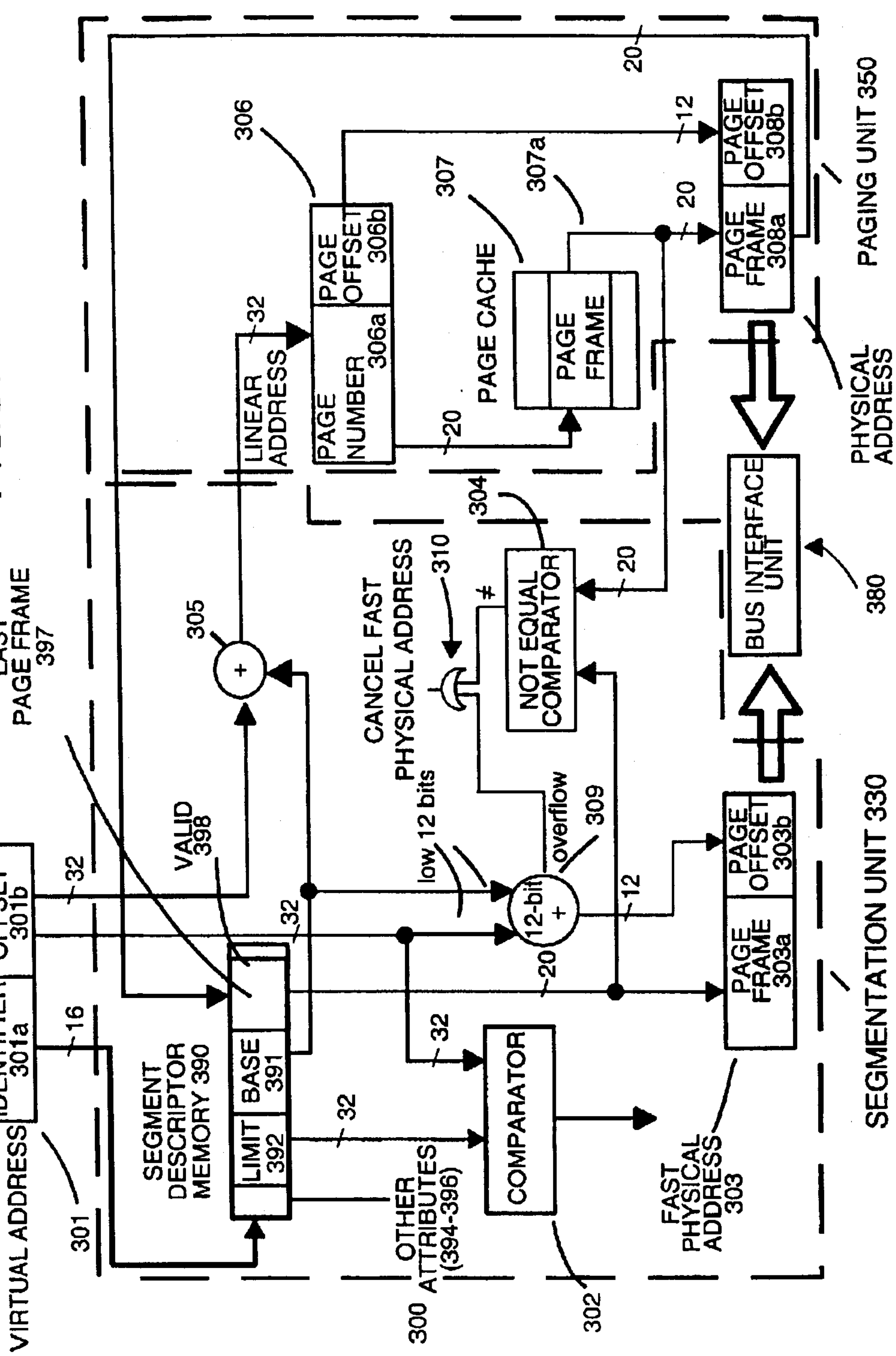
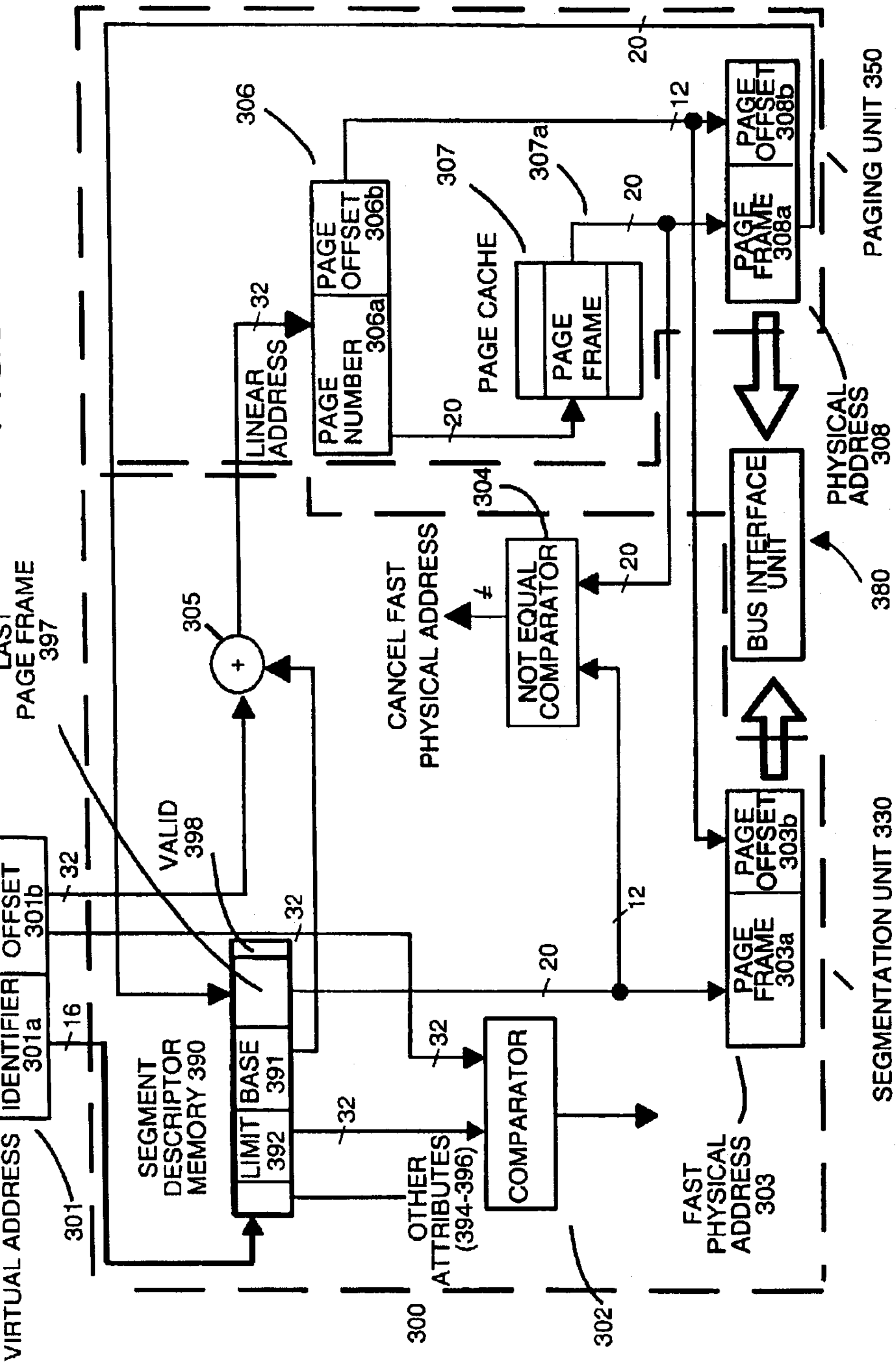

VIRTUAL ADDRESS
301
SEGMENT IDENTIFIER 301a
SEGMENT OFFSET 301b
16
32
LAST PAGE FRAME 397
VALID 398
SEGMENT DESCRIPTOR MEMORY 390
BASE 391
LIMIT 392
OTHER ATTRIBUTES (394-396)
300
COMPARATOR
302
305
LINEAR ADDRESS
PAGE NUMBER 306a
PAGE OFFSET 306b
306
CANCEL FAST PHYSICAL ADDRESS
310
low 12 bits
12-bit
overflow
309
NOT EQUAL COMPARATOR
≠
304
PAGE CACHE
PAGE FRAME
307
307a
PAGE FRAME 308a
PAGE OFFSET 308b
PHYSICAL ADDRESS 308
PAGING UNIT 350
BUS INTERFACE UNIT
380
PAGE FRAME 303a
PAGE OFFSET 303b
FAST PHYSICAL ADDRESS 303
SEGMENTATION UNIT 330
12
20

FIGURE 3C


VIRTUAL ADDRESS
301
SEGMENT IDENTIFIER 301a
SEGMENT OFFSET 301b
16
32
LAST PAGE FRAME 397
VALID 398
SEGMENT DESCRIPTOR MEMORY 390
BASE 391
LIMIT 392
OTHER ATTRIBUTES (394-396)
COMPARATOR
300
302
305
+
LINEAR ADDRESS
PAGE NUMBER 306a
PAGE OFFSET 306b
306
PAGE CACHE
PAGE FRAME
307
307a
304
CANCEL FAST PHYSICAL ADDRESS
≠
NOT EQUAL COMPARATOR
12
20
FAST PHYSICAL ADDRESS 303
PAGE FRAME 303a
PAGE OFFSET 303b
SEGMENTATION UNIT 330
BUS INTERFACE UNIT
380
PHYSICAL ADDRESS 308
PAGE FRAME 308a
PAGE OFFSET 308b
PAGING UNIT 350

SPECULATIVE ADDRESS TRANSLATION FOR PROCESSOR USING SEGMENTATION AND OPTIONAL PAGING

RELATED APPLICATION DATA

This application is a continuation of application Ser. No. 09/757,439 filed Jan. 10, 2001, now U.S. Pat. No. 6,430,668, which is a continuation of Ser. No. 08/905,356 filed Aug. 4, 1997, now U.S. Pat. No. 6,226,733, which in turn is a continuation of Ser. No. 08/458,479 filed on Jun. 2, 1995, now U.S. Pat. No. 5,895,503.

FIELD OF THE INVENTION

The invention relates to the field of address translation for memory management in a computer system.

BACKGROUND OF THE INVENTION

Advanced computer hardware systems operate with complex computer software programs. Computer system designers typically separate the virtual address space, the address space used by programmers in their development of software, and the physical address space, the address space used by the computer system. This separation allows programmers to think in terms of their conceptual models, and to design computer software programs without reference to specific hardware implementations. During the actual execution of programs by the computer system, however, these separate addresses must be reconciled by translating software program virtual addresses into actual physical addresses that can be accessed in a computer memory subsystem.

There are many well known approaches for address translation in the memory management mechanism of a computer system. These approaches fall into basically two major categories: those which map the smaller virtual (sometimes called logical, symbolic or user) addresses onto larger physical or real memory addresses, and those which map larger virtual addresses onto smaller physical memory. Translation mechanisms of the former category are employed typically in minicomputers in which relatively small address fields (e.g.: 16 bit addresses) are mapped onto larger real memory. Translation mechanisms of the second category are used typically in microprocessors, workstations and mainframes. Within each of these categories, segmentation only, paging only, and a combination of segmentation and paging are well known for accomplishing the translation process.

The present invention is primarily directed to address translation mechanisms where larger virtual addresses are mapped onto smaller physical addresses, and further to systems where segmentation and optional paging is employed.

In a segmentation portion of an address translation system, the address space of a user program (or programs cooperatively operating as processes or tasks), is regarded as a collection of segments which have common high-level properties, such as code, data, stack, etc. The segmented address space is referenced by a 2-tuple, known as a virtual address, consisting of the following fields: <<s>,<d>>, where <s> refers to a segment number (also called identifier or locator), and <d> refers to a displacement or offset, such as a byte displacement or offset, within the segment identified by the segment number. The virtual address <17,421>, for example, refers to the 421st byte in segment 17. The segmentation portion of the address translation mechanism,

using information created by the operating system of the computer system, translates the virtual address into a linear address in a linear address space.

In a paging portion of an address translation system, a linear (or intermediate) address space consists of a group of pages. Each page is the same size (i.e. it contains the same number of addresses in the linear space). The linear address space is mapped onto a multiple of these pages, commonly, by considering the linear address space as the 2-tuple consisting of the following fields: <<page number >,<page offset >>. The page number (or page frame number) determines which linear page is referenced. The page offset is the offset or displacement, typically a byte offset, within the selected page.

In a paged system, the real (physical) memory of a computer is conceptually divided into a number of page frames, each page frame capable of holding a single page. Individual pages in the real memory are then located by the address translation mechanism by using one or more page tables created for, and maintained by, the operating system. These page tables are a mapping from a page number to a page frame. A specific page may or may not be present in the real memory at any point in time.

Address translation mechanisms which employ both segmentation and paging are well known in the art. There are two common subcategories within this area of virtual address translation schemes: address translation in which paging is an integral part of the segmentation mechanism; and, address translation in which paging is independent from segmentation.

In prior art address translation mechanisms where paging is an integral part of the segmentation mechanism, the page translation can proceed in parallel with the segment translation since segments must start at page boundaries and are fixed at an integer number of pages. The segment number typically identifies a specific page table and the segment offset identifies a page number (through the page table) and an offset within that page. While this mechanism has the advantage of speed (since the steps can proceed in parallel) it is not flexible (each segment must start at a fixed page boundary) and is not optimal from a space perspective (e.g. an integer number of pages must be used, even when the segment may only spill over to a fraction of another page).

In prior art address translation mechanisms where paging is independent from segmentation, page translation generally cannot proceed until an intermediate, or linear, address is first calculated by the segmentation mechanism. The resultant linear address is then mapped onto a specific page number and an offset within the page by the paging mechanism. The page number identifies a page frame through a page table, and the offset identifies the offset within that page. In such mechanisms, multiple segments can be allocated into a single page, a single segment can comprise multiple pages, or a combination of the above, since segments are allowed to start on any byte boundary, and have any byte length. Thus, in these systems, while there is flexibility in terms of the segment/page relationship, this flexibility comes at a cost of decreased address translation speed.

Certain prior art mechanisms where segmentation is independent from paging allow for optional paging. The segmentation step is always applied, but the paging step is either performed or not performed as selected by the operating system. These mechanisms typically allow for backward compatibility with systems in which segmentation was present, but paging was not included.

Typical of the prior art known to the Applicant in which paging is integral to segmentation is the Multics virtual memory, developed by Honeywell and described by the book, "The Multics System", by Elliott Organick. Typical of the prior art known to the Applicant in which optional paging is independent from segmentation is that described in U.S. Pat. No. 5,321,836 assigned to the Intel Corporation, and that described in the Honeywell DPS-8 Assembly Instructions Manual. Furthermore, U.S. Pat. No. 4,084,225 assigned to the Sperry Rand Corporation contains a detailed discussion of general segmentation and paging techniques, and presents a detailed overview of the problems of virtual address translation.

Accordingly, a key limitation of the above prior art methods and implementations where segmentation is independent from paging is that the linear address must be fully calculated by the segmentation mechanism each time before the page translation can take place for each new virtual address. Only subsequent to the linear address calculation, can page translation take place. In high performance computer systems computer systems, this typically takes two full or more machine cycles and is performed on each memory reference. This additional overhead often can reduce the overall performance of the system significantly.

SUMMARY OF THE INVENTION

An object of the present invention, therefore, is to provide the speed performance advantages of integral segmentation and paging and, at the same time, provide the space compaction and compatibility advantages of separate segmentation and paging.

A further object of the present invention is to provide a virtual address translation mechanism which architecturally provides for accelerating references to main memory in a computer system which employs segmentation, or which employs both segmentation and optional paging.

Another object of the present invention is to provide additional caching of page information in a vital address translation scheme.

An further object of the present invention is to provide a virtual address translation mechanism which reduces the number of references required to ensure memory access.

According to the present invention, a segmentation unit converts a virtual address consisting of a segment identifier and a segment offset into a linear address. The segmentation unit includes a segment descriptor memory, which is selectable by the segment identifier. The entry pointed to by the segment identifier contains linear address information relating to the specific segment (i.e., linear address information describing the base of the segment referred to by the segment identifier, linear address information describing the limit of the segment referred to by the segment identifier, etc.) as well as physical address information pertaining to the segment—such as the page base of at least one of the pages represented by said segment.

In the above embodiment, unlike prior art systems, both segmentation and paging information are kept in the segmentation unit portion of the address translation system. The caching of this page information in the segmentation unit permits the address translation process to occur at much higher speed than in prior art systems, since the physical address information can be generated without having to perform a linear to physical address mapping in a separate paging unit.

The page base information stored in the segmentation unit is derived from the page frame known from the immediately prior in time address translation on a segment-by-segment basis. In order to complete the full physical address translation (i.e., a page frame number and page offset), the segmentation unit combines the page frame from the segment descriptor memory with the page offset field, and may store this result in a segmentation unit memory, which can be a memory table, or a register, or alternatively, it may generate the full physical address on demand.

This fast physical address generated by the segmentation unit based on the virtual address and prior page information can be used by a bus interface to access a physical location in the computer memory subsystem, even before the paging unit has completed its translation of the linear address into a page frame and page offset. Thus, fewer steps and references are required to create a memory access. Consequently, the address translation step occurs significantly faster. Since address translation occurs in a predominant number of instructions, overall system performance is improved.

The memory access is permitted to proceed to completion unless a comparison of the physical address information generated by the paging unit with the fast physical address generated by the segmentation unit shows that the page frame information of the segmentation unit is incorrect.

In alternative embodiments, the segmentation unit either generates the page offset by itself (by adding the lower portion of the segment offset and the segment base address) or receives it directly from the paging unit.

In further alternate embodiments, the incoming segment offset portion of the virtual address may be presented to the segmentation unit as components. The segmentation unit then combines these components in a typical base-plus-offset step using a conventional multiple input (typically 3-input) adder well known in the prior art.

As shown herein in the described invention, the segment descriptor memory may be a single register, a plurality of registers, a cache, or a combination of cache and register configurations.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block of a typical prior art virtual address translation mechanism using segmentation and independent paging.

FIG. 2A is a detailed diagram of a typical segment descriptor register of the prior art.

FIG. 2B is a detailed diagram of an embodiment of the present invention, including a portion of a segment descriptor memory used for storing physical address information;

FIG. 3A is a block diagram of an embodiment of the present invention employing segmentation and optional paging, and showing the overall structure and data paths used when paging is disabled during an address translation;

FIG. 3B is a block diagram of the embodiment of FIG. 3A showing the overall structure and data paths used when paging is enabled during an address translation;

FIG. 3C is a block diagram of another embodiment of the present invention, showing an alternative circuit for generating the fast physical address information.

DETAILED DESCRIPTION OF THE MENTION

General Discussion of Paging & Segmentation

The present invention provides for improved virtual address translation in a computer system. The preferred embodiment employs the invention in a single-chip microprocessor system, however, it is well understood that such a virtual address translation system could be implemented in

multiple die and chip configurations without departing from the spirit or claims of the present invention.

Before embarking on a specific discussion of the present invention, however, a brief explanation of the general principles of segmentation and paging follows in order to provide additional background information, and so that the teachings of the present invention may be understood in a proper context.

Referring to FIG. **1**, a typical prior art virtual address translation mechanism **100** using both segmentation and, optionally, paging in a computer system is shown. As described in this figure, a data path within the microprocessor transmits a virtual address **101**, consisting of segment identifier **101***a* and a segment offset **101***b*, to segmentation unit **130**. Segments are defined by segment descriptor entries in at least one segment descriptor table or segment descriptor segment (not shown). Segment descriptor tables are created and managed by the operating system of the computer system, and are usually located in the memory subsystem. Segment descriptor entries are utilized in the CPU of the computer system by loading them into segment descriptor register **190** or a segment descriptor cache (not shown); the segment descriptor register/cache is usually internal to the CPU, and thus more quickly accessible by the translation unit.

In the paging unit **150**, pages are defined by a page table or multiple page tables (not shown), also created and managed by the operating system; again, these tables are also typically located in a memory subsystem. All or a portion of each page table can be loaded into a page cache **107** (within the CPU, sometimes called a translation look-aside buffer) to accelerate page references.

In operation, the segmentation unit **130** first translates a virtual address to a linear address and then (except in the case when optional paging is disabled) paging unit **150** translates the linear address into a real (or physical) memory address.

Typically (as in an x86 microprocessor) the segmentation unit translates a 48-bit virtual address **101** consisting of a 16-bit segment identifier (<s>) **101***a* and a 32-bit displacement within that segment (<d>) **101***b* to a 32-bit linear (intermediate) address **106**. The 16-bit segment identifier **101***a* uniquely identifies a specific segment; this identifier is used to access an entry in a segment descriptor table (not shown). In the prior art, this segment descriptor entry contains a base address of the segment **191**, the limit of the segment **192**, and other attribute information described further below. The segment descriptor entry is usually loaded into a segment descriptor register **190**.

Using adder **105**, the segmentation unit adds the segment base **191** of the segment to the 32-bit segment offset **101***b* in the virtual address to obtain a 32-bit linear address. The 32-bit segment offset **10***b* in the virtual address is also compared against the segment limit **192**, and the type of the access is checked against the segment attributes. A fault is generated and the addressing process is aborted if the 32-bit segment offset is outside the segment limit, or if the type of the access is not allowed by the segment attributes.

The resulting linear address **106** can be treated as an offset within a linear address space; and in the commonly implemented schemes of the prior art, these offsets are frequently byte offsets. When optional paging is disabled, the linear address **106** is exactly the real or physical memory address **108**. When optional paging is enabled, the linear address is treated as a 2- or 3-tuple depending on whether the paging unit **150** utilizes one or two level page tables.

In the 2-tuple case shown in FIG. **1** which represents single level paging, the linear address, <<p>,<pd>> is divided into a page number field <p> **106***a*, and a page displacement page offset) field within that page (<pd>) **106***b*. In the 3-tuple case (not shown) <<dp>,<p>,<pd>>, the linear address is divided into a page directory field (<dp>), a page number field <p> and a page displacement field <pd>. The page directory field indexes a page directory to locate a page table (not shown). The page number field indexes a page table to locate the page frame in real memory corresponding to the page number, and the page displacement field locates a byte within the selected page frame. Thus, paging unit **150** translates the 32-bit linear address **106** from the segmentation unit **130** to a 32-bit real (physical) address **108** using one or two level page tables using techniques which are well known in the art.

In all of the above prior art embodiments where segmentation is independent from paging, the segment descriptor table or tables of the virtual address translator are physically and logically separate from the page tables used to perform the described page translation. There is no paging information in the segment descriptor tables and, conversely, there is no segmentation information in the page tables.

This can be seen in FIG. **2A**. In this figure, a typical prior art segment descriptor entry **200**, is shown as it is typically used in a segment descriptor table or segment descriptor register associated with a segmentation unit. As can be seen there, segment descriptor **200** includes information on the segment base **201**, the segment limit **202**, whether the segment is present (P) **203** in memory, the descriptor privilege level (DPL) **204**, whether the segment belongs to a user or to the system (S) **205**, and the segment type **206** (code, data, stack, etc.)

For additional discussions pertaining to the prior art in segmentation, paging, segment descriptor tables, and page tables, the reader is directed to the references U.S. Pat. Nos. 5,408,626, 5,321,836, 4,084,225, which are expressly incorporated by reference herein.

Improved Segmentation Unit Using Paging Information

As shown in the immediate prior art, the paging and segmentation units (circuits) are completely separate and independent. Since the two units perform their translation sequentially, that is, the segment translation must precede the page translation to generate the linear address, high performance computer systems, such as those employing superscalar and superpipelined techniques, can suffer performance penalties. In some cases, it is even likely that the virtual address translation could fall into the systems' "critical path". The "critical path" is a well-known characteristic of a computer system and is typically considered as the longest (in terms of gate delay) path required to complete a primitive operation of the system.

Accordingly, if the virtual address translation is in the critical path, the delays associated with this translation could be significant in overall system performance. With the recognition of this consideration, the present invention includes page information in the segment translation process. The present invention recognizes the potential performance penalty of the prior art and alleviates it by storing paging information in the segmentation unit obtained from a paging unit in previous linear-to-real address translations.

As can be seen in FIG. **2B**, the present invention extends the segment descriptor entries of the prior art with a segment entry **290** having two additional fields: a LAST PAGE FRAME field **297** and a VALID field **298**. The LAST PAGE FRAME field **297** is used to hold the high-order 20 bits (i.e.: the page frame) of the real (physical) memory address of the last physical address generated using the specified segment identifier. The VALID field **298** is a 1-bit field, and indicates

whether or not the LAST PAGE FRAME field **297** is valid. The remaining fields **291–296** perform the same function as comparable fields **201–206** respectively described above in connection with FIG. 2A.

Segment descriptor tables (not shown) can be located in a memory subsystem, using any of the techniques well-known in the art. As is also known in the art, it is possible to speed up address translation within the segmentation unit by using a small cache, such as one or more registers, or associative memory. The present invention makes use of such a cache to store segment entries **290** shown above. Unlike the prior art, however, the segment entries **290** of the present invention each contain information describing recent physical address information for the specified segment. Accordingly, this information can be used by a circuit portion of the segmentation unit to generate a new physical address without going through the linear to physical mapping process typically associated with a paging unit.

While in some instances the physical address information may change between two time-sequential virtual addresses to the same segment (and thus, a complete translation is required by both the segmentation and paging units), in the majority of cases the page frame information will remain the same. Thus, the present invention affords a significant speed advantage over the prior art, because in the majority of cases a complete virtual-linear-physical address translation is not required before a memory access is generated.

Embodiment with Segmentation & Optional Paging/Paging Disabled

Referring to FIG. 3A, the advantage of using this new information in segment entry **290** in a segmentation unit or segmentation circuit is apparent from a review of the operation of an address translation. In this figure, a paging unit (or paging circuit) is disabled, as for example might occur only when a processor is used in a real mode of operation, rather than a protected mode of operation.

In a preferred embodiment, the present invention employs a segment descriptor memory comprising at least one, and preferably many, segment descriptor registers **390**, which are identical in every respect to the segment descriptor register described above in connection with FIG. 2B. These segment descriptor registers are loaded from conventional segment descriptor tables or segment descriptor segments which are well known in the art. Each segment descriptor register **390** is loaded by the CPU before it can be used to reference physical memory. Segment descriptor register **390** can be loaded by the operating system or can be loaded by application programs. Certain instructions of the CPU are dedicated to loading segment descriptor registers, for example, the "LDS" instruction, "Load Pointer to DS Register". Loading by the operating system, or execution of instructions of this type, causes a base **391**, limit **392**, descriptor privilege level **394**, system/user indicator **395**, and type **396** to be loaded from segment tables or segment descriptor segments as in the prior art. The three remaining fields are present **393**, LAST PAGE FRAME **397** and VALID **398**. When a segment descriptor register **390** is loaded, present **393** is set to 1, indicating that the segment descriptor register **390** contents are present; the valid field **398** is set to 0, indicating that the last page frame number field **397** is not valid; and the LAST PAGE FRAME field **397** is not set, or may be set to 0.

After the loading of a segment descriptor register **390**, instructions of the CPU may make references to virtual memory; if a segment descriptor register is referenced before it is loaded, as indicated by present field **393** set to 0, a fault occurs and the reference to the segment descriptor register is aborted.

As explained above, the CPU makes references to virtual memory by specifying a 48-bit virtual address, consisting of a 16-bit segment identifier **301*a*** and a 32-bit segment offset **301*b***. A data path within the CPU traits virtual address **301** to the address translation mechanism **300**.

Segment descriptor memory **390** is indexed by segment number, so each entry in this memory containing data characteristics (i.e., base, access rights, limit) of a specific segment is selectable by the segment identifier from the vital address. Assuming this is the first reference to physical memory specifying a newly loaded segment descriptor register, since the VALID bit **398** is set to false, a prior art virtual address translation takes place. This involves, among other things, as explained earlier, various validity checks (including checking attributes **394–396**, segment limit checking using comparator **302** and potentially others), and using adder **305** to add the segment descriptor's base address **391** to the segment offset **301*b*** to calculate a linear address **306**.

While the implementation in the embodiment of FIG. 3A shows the addition of the base address **391** to the segment offset **301*b*** using adder **305** to generate the linear address **306**, it will be understood by those skilled in the art that this specific implementation of the virtual to linear address translation is not the only implementation of the present invention. In other implementations, the segment offset **301*b*** might consist of one or more separate components. Different combinations of one or more of these components might be combined using well known techniques to form a linear address, such as one utilizing a three-input adder. The use of these components is discussed, for example, in U.S. Pat. No. 5,408,626, and that description is incorporated by reference herein.

As is well known, in this embodiment where paging is disabled, linear address **306** is also a physical address which can be used as the physical address **308**. Memory access control operations are not shown explicitly since they are only ancillary to the present invention, and are well described in the prior art. In general, however, a bus interface unit **380** is typically responsible for interactions with the real (physical) memory subsystem. The memory subsystem of a computer system employing the present invention preferably has timing and address and data bus transaction details which are desirably isolated from the CPU and address translation mechanism. The bus interface unit **380** is responsible for this isolation, and can be one of many conventional bus interface units of the prior art.

In the present invention, bus interface unit **380** receives the real memory address **308** from address translation mechanism **300** and coordinates with the real memory subsystem to provide data, in the case of a memory read request, or to store data, in the case of a memory write request. The real memory subsystem may comprise a hierarchy of real memory devices, such as a combination of data caches and dynamic RAM, and may have timing dependencies and characteristics which are isolated from the CPU and address translation mechanism **300** of the computer system by the bus interface unit **380**.

Simultaneous with the first memory reference using the calculated physical address **308**, the LAST PAGE FRAME field of the selected segment descriptor register **390** is loaded with the high-order 20 bits of the physical address, i.e.: the physical page frame, and the VALID bit is set to indicate a valid state. This paging information will now be used in a next virtual address translation.

Accordingly, when a next, new virtual address **301** is to be translated, the entry selected from segment descriptor

memory 390 will likely contain the correct physical frame page number (in the LAST PAGE FRAME field 397). Thus, in most cases, the base physical address in memory for the next, new referenced virtual address will also be known from a previous translation.

The first step of the virtual address translation, therefore, is to determine if a FAST PHYSICAL ADDRESS 303 can be used to begin a fast physical memory reference. Adder 309, a 12-bit adder, adds the low-order 12 bits of the segment offset 301*b* of virtual address 301 to the low-order 12-bits of base 391 of the segment entry in segment descriptor register 390 referenced by the segment identifier 301*a*. This addition results in a page offset 303*b*. In parallel with adder 309, 32-bit adder 305 begins a fill 32-bit add of segment base 391 and segment offset 301*b*, to begin producing the linear address; however, this full 32-bit add will obviously require more time. In the preferred embodiment, adder 309 is a separate 12-bit adder, however, it should be noted that adder 309 also could be implemented as the low order 12-bits of 32-bit adder 305.

Simultaneous with the beginning of these two operations, VALID bit 398 is inspected. If VALID bit 398 is set to 1, as soon as 12-bit adder 309 has completed, 20-bit LAST PAGE FRAME 397 is concatenated with the result of adder 309 to produce FAST PHYSICAL ADDRESS 303, consisting of a page frame number 303*a*, and page offset 303*b*. FAST PHYSICAL ADDRESS 303 then can be used to tentatively begin a reference to the physical memory. It should be understood that the FAST PHYSICAL ADDRESS 303 transmitted to bus interface unit 380 could also be stored in a register or other suitable memory storage within the CPU.

In parallel with the fast memory reference, limit field 392 is compared to the segment offset 301*b* of the virtual address by comparator 302. If the offset in the virtual address is greater than the limit, a limit fault is generated, and vital address translation is aborted.

Also in parallel with the fast memory reference, adder 305 completes the addition of base 391 to the segment offset field 301*b* of virtual address to produce linear address (in this case physical address also) 306. When this calculation is completed, the page frame number 308*a* of physical address 308 is compared to LAST PAGE FRAME 397 by Not Equal Comparator 304. If page frame 308*a* is unequal to the LAST PAGE FRAME 397, or if 12-bit Adder 309 overflowed (as indicated by a logic "1" at OR gate 310), the fast memory reference is canceled, and the linear address 306, which is equal to the physical address 308, is used to begin a normal memory reference. If page frame 308*a* is equal to LAST PAGE FRAME 397, and 12-bit Adder 309 did not overflow (the combination indicated by a logic "0" at the output of OR gate 310), the fast memory reference is allowed to fully proceed to completion.

After any fast memory reference which is cancelled by the CANCEL FAST PHYSICAL ADDRESS signal output of OR gate 310, page frame 308*a* is loaded into the LAST PAGE FRAME 397 in the segment descriptor memory 390 for subsequent memory references.

Depending on the particular design desired, it should also be noted that writes to the memory, or reads which cause faults using FAST PHYSICAL ADDRESS 303 may be pended since the FAST PHYSICAL ADDRESS 303 may prove to be invalid.

Accordingly, it can be seen that the parallel physical address calculation undertaken by the improved segmentation unit of the present invention generates a faster physical memory access than possible with prior art systems.

Embodiment with Segmentation & Paging/Paging Enabled

The present invention can also be used with address translation units using paging enabled, as can be seen in the embodiments of FIGS. 3B and 3C.

In the embodiment of FIG. 3B, the same segmentation unit structure 300 as that shown in FIG. 3A is used, and the operation of segmentation unit 300 is identical to that already explained above. As before, segment descriptor memory (registers) 390 are loaded from conventional segment descriptor tables or segment descriptor segments, using one or more of the procedures described above. First, the base 391 limit 392 descriptor privilege level 394, system/user indicator 395, and type 396 are loaded from segment tables or segment descriptor segments as explained earlier. When segment descriptor register 390 is loaded, present 393 is set to 1, indicating that the segment descriptor register 390 contents are present; the valid field 398 is set to 0, indicating that the last page frame number field 397 is not valid; and the LAST PAGE FRAME field 397 is not set, or may be set to 0.

As explained above, after the loading of a segment descriptor register 390, instructions of the CPU may make references to virtual memory; if a segment descriptor register is referenced before it is loaded, as indicated by present field 393 set to 0, a fault occurs and the reference to the segment descriptor register is aborted.

As further explained above, the 48 bit virtual address 301 (consisting of a 16 bit segment identifier 301*a* and a 32 bit segment offset 301*b*) is transmitted by a data path to segmentation unit 300, and an index into segment descriptor memory 390 is performed to locate the specific segment descriptor for the segment pointed to by segment identifier 301*a*. Assuming this is the first reference to physical memory specifying a newly loaded segment descriptor register, since the VALID bit is set to false, a prior art virtual address translation takes place. This involves, among other things, as explained earlier, various validity checks (including checking attributes 394–396, segment limit checking using comparator 302 and potentially others), and using adder 305 to add the segment descriptor's base address 391 to the segment offset 301*b* to calculate a linear address 306.

As is well known, in this configuration where paging is enabled, linear address 306 must undergo a further translation by paging unit 350 to obtain the physical address 308 in the memory subsystem. In the preferred embodiment of the invention, looking first at FIG. 3B, the output of adder 305 will be a 32-bit linear address, corresponding to a 20-bit page number 306*a* and a 12-bit page offset 306*b*. Typically, the page number 306*a* is then indexed into a page descriptor table (not shown) to locate the appropriate page frame base physical address in memory. These page descriptor tables are set up by the operating system of the CPU using methods and structures well known in the art, and they contain, among other things, the base physical address of each page frame, access attributes, etc.

However, in most systems, including the present invention, a page cache 307 is used in order to hold the physical base addresses of the most recently used page frames. This cache can take the form of a table, associative cache, or other suitable high speed structure well known in the art. Thus, page number 306*a* is used to access page data (including physical base addresses for page frames) in an entry in page cache 307.

If page cache 307 hits, two thins happen first, a 20-bit PAGE FRAME 307*a* (the page frame in physical memory) replaces the high-order 20 bits (page number 306*a*) of the

linear address **306**, and, when concatenated with the page offset **306*b*** results in a real (physical) address **308**, which is used to perform a memory access through bus interface unit **380** along the lines explained above. Second, newly generated page frame **308*a*** is also stored in segment descriptor memory **390** in the selected LAST PAGE FRAME field **397** to be used for a fast access in the next address translation. When LAST PAGE FRAME field **397** is stored, selected VALID bit **398** is set to 1 to indicate that LAST PAGE FRAME **397** is valid for use.

In the event of a page cache miss, the appropriate page frame number **308*a*** is located (using standard, well-known techniques) to generate physical address **308**, and is also loaded into segment descriptor memory **390** in the selected LAST PAGE FRAME field **397**. The selected VALID bit **398** is also set to indicate a valid state. Thus, there is paging information in the segmentation unit that will now be used in the nest virtual address translation.

When a next, new virtual address **301** is to be translated, the segment identifier **301*a*** will likely be the same as that of a previously translated virtual address, and the entry selected from segment descriptor memory **390** will also likely contain the correct physical frame (in LAST PAGE FRAME field **397**) from the previous translation. As with the above embodiment, one or more registers, or a cache may be used for the segment descriptor memory **390**.

The first step then determines if a FAST PHYSICAL ADDRESS **303** can be used to begin a fast physical memory reference. Adder **309**, a 12-bit adder, adds the low-order 12 bits of the segment offset **301*b*** of oval address **301** to the low-order 12-bits of base **391** of the segment entry in segment descriptor register **390** referenced by the segment identifier **301*a***. This addition results in a page offset **303*b***. In parallel with adder **309**, 32-bit adder **305** begins a full 32-bit add of segment base **301** and segment offset **301*b***, to begin producing the linear address; however, this full 32-bit add will obviously require more time. In the preferred embodiment, adder **309** is a separate 12-bit adder, however, it should be noted that adder **309** also could be implemented as the low order 12-bits of 32-bit adder **305**.

Simultaneous with these beginning of these two operations, VALID bit **398** is inspected. If VALID bit **398** is set to 1, as soon as 12-bit adder **309** has completed, 20-bit LAST PAGE FRAME **397** is concatenated with the result of adder **309** to produce FAST PHYSICAL ADDRESS **303**, consisting of a page frame number **303*a***, and page offset **303*b***. FAST PHYSICAL ADDRESS **303** then can be used to tentatively begin a reference to the physical memory. Again, it should be understood that the FAST PHYSICAL ADDRESS **303** transmitted to bus interface unit **380** could also be stored in a register or other suitable memory storage within the CPU.

As before, limit field **302** is compared to the segment offset **301*b*** of the virtual address by comparator **302**. If the offset in the virtual address is greater than the limit, a limit fault is generated, and virtual address translation is aborted.

This new virtual address is also translated by paging unit **350** in the same manner as was done for the previous virtual address. If page cache **307** hits based on the page number **306*a***, two things happen: first, a 20-bit PAGE FRAME **307*a*** (the page frame in physical memory) replaces the high-order 20 bits (page number **306*a***) of the linear address **306**, and, when concatenated with the page offset **306*b*** results in a physical address **308**. This real address may or may not be used, depending on the result of the following: in parallel with the aforementioned concatenation, the PAGE FRAME **307*a***, is compared to LAST PAGE FRAME **397** from the segment descriptor memory **390** by Not Equal Comparator **304**. The result of Not Equal Comparator (that is, the Not Equal condition) is logically ORed with the overflow of 12-bit adder **309** by OR gate **310**. If the output of OR gate **310** is true (i.e. CANCEL FAST PHYSICAL ADDRESS is equal to binary one), or if PAGE CACHE **307** indicates a miss condition, the fast memory reference previously begun is canceled, since the real memory reference started is an invalid reference. Otherwise, the fast memory reference started is allowed to fully proceed to completion, since it is a valid real memory reference.

If CANCEL FAST PHYSICAL ADDRESS is logic true, it can be true for one of two, or both reasons. In the case that Or gate **310** is true, but page cache **307** indicates a hit condition, physical address **308** is instead used to start a normal memory reference. This situation is indicative of a situation where LAST PAGE FRAME **397** is different from the page frame **308*a*** of the current reference.

In the case that page cache **307** did not indicate a hit, a page table reference through the page descriptor table is required and virtual address translation proceeds as in the prior art. The page frame **308*a*** information is again stored in the LAST PAGE FRAME field **397** in the segment descriptor memory **390** for the next translation.

Also, after any fast memory reference which is canceled by the CANCEL FAST PHYSICAL ADDRESS signal output of OR gate **310**, page frame number **308*a*** is loaded into the LAST PAGE FRAME **397** in the segment descriptor memory **390** for subsequent memory references.

Depending on the particular design desired, it should also be noted that in this embodiment also, writes to the memory, or reads which cause faults using FAST PHYSICAL ADDRESS **303** may be pended since the FAST PHYSICAL ADDRESS **303** may prove to be invalid.

The alternative embodiment shown in FIG. **3C** is identical in structure and operation to the embodiment of FIG. **3B**, with the exception that the 12-bit Adder **309** is not employed. In this embodiment, the segmentation unit **330** does not create the lower portion (page offset **303*a***) of the fast physical address in this manner. Instead, the page offset **306*a*** resulting from 32-bit adder **305** is used.

It can be seen that the present invention has particular relevance to computers using sequential type of segmentation and paging translation, such as the X86 family of processors produced by the Intel Corporation (including the Intel 80386, Intel 80486 and the Intel Pentium Processor), other X86 processors manufactured by the NexGen Corporation, Advanced Micro Devices, Texas Instruments, International Business Machines, Cyrix Corporation, and certain prior art computers made by Honeywell. These processors are provided by way of example, only, and it will be understood by those skilled in the art that the present invention has special applicability to any computer system where software executing on the processors is characterized by dynamic execution of instructions in programs in such a way that the virtual addresses are generally logically and physically located near previous virtual addresses.

The present invention recognizes this characteristic, employing acceleration techniques for translating virtual to real addresses. In particular, the present invention utilizes any of the commonly known storage structures (specific examples include high speed registers and/or caches) to store previous address translation information, and to make this previous address translation information available to the system whenever the next subsequent reference relies on the same information. In this way, the system can utilize the previously stored information from the high speed storage to

begin real memory references, rather than be forced to execute a more time consuming translation of this same information, as was typically done in the prior art.

As will be apparent to those skilled in the art, there are other specific circuits and structures beyond and/or in addition to those explicitly described herein which will serve to implement the translation mechanism of the present invention. Finally, although the above description enables the specific embodiment described herein, these specifics are not intended to restrict the invention, which should only be limited as defined by the following claims.

I claim:

1. A method of generating a speculative memory address from a virtual address having both a segment identifier and a segment offset in a computer system employing both segmentation and optional independent paging, the method including the steps of:

(a) converting a portion of the virtual address into a partial linear address; and

(b) combining the partial linear address with physical address information obtained from a prior memory address generation to generate the speculative memory address.

2. The method of claim 1, wherein the speculative memory address is used to initiate a speculative memory access.

3. The method of claim 2 wherein the speculative memory access is to a cache memory.

4. A method of generating a speculative memory access to a memory associated with a microprocessor based on translating a virtual address having both a segment identifier and a segment offset in a computer system employing both segmentation and optional independent paging, the method including the steps of:

(a) performing a first complete translation of a first virtual address to generate a first complete physical address from a complete linear address, said first complete physical address including a first physical page frame;

(b) storing the first physical page frame in a first cache within the microprocessor;

(c) subsequently converting at least a portion of a second virtual address into a partial linear address in the microprocessor while a second complete translation is occurring for the second virtual address to generate a complete second physical address from a complete second linear address, said second complete physical address including a second physical page frame;

(d) using the partial linear address to perform a first fast memory access to a data cache in parallel with said second complete translation;

(e) comparing the first physical page frame with said second physical page frame;

(f) performing a second normal memory access when said first physical page frame is not equal to said second physical page frame.

5. The method of claim 4, wherein said complete second linear address is 32 bits long, and said partial linear address consists of low order bits of said complete second linear address.

6. The method of claim 4, wherein writes to said data cache are pended until said first memory access is determined to be valid.

7. A system for performing page address translation for a virtual address to physical address translation within a processor that employs both segmentation and optional independent paging, said system comprising:

a) a linear address generator adapted to calculate a calculated linear address based on processing the entire virtual address;

b) a physical address generator, coupled to the linear address generator, adapted to generate an actual physical page frame based on processing all of said calculated linear address; and

c) a second memory storing at least one speculative physical page frame associated with the virtual address;

wherein the speculative physical page frame is compared to the actual physical page frame to determine if the speculative physical page frame is valid.

8. The system of claim 7, wherein said second memory also includes a field for indicating whether said at least one speculative physical page frame is valid.

9. The system of claim 7, wherein said second memory includes a plurality of speculative physical page frames corresponding to a plurality of previously translated virtual addresses.

10. A system for performing address translation for a virtual address to physical address translation within a processor that employs both segmentation and optional independent paging, said system comprising:

a) a linear address generator adapted to calculate a first calculated linear address based on processing an entire first virtual address;

b) a physical address generator, coupled to the linear address generator, adapted to generate an actual first physical address based on processing all of said calculated linear address; and

c) an adder for calculating a first partial calculated linear address based on processing only a portion of said entire first virtual address;

wherein said first partial calculated linear address further is used by the system to initiate a fast memory access to a data cache before said actual first physical address is generated by the physical address generator.

11. The system of claim 10, wherein said adder is separate from a linear address generator adder used to compute said first calculated linear address.

12. The system of claim 10, wherein said first partial calculated linear address is calculated by a lower order portion of a linear address generator adder that is also used to compute said first calculated linear address.

13. A system for performing address translation for a virtual address to physical address translation within a processor that employs both segmentation and optional independent paging, said system comprising:

a) a linear address generator adapted to calculate a first calculated linear address based on processing an entire first virtual address;

b) a physical address generator, coupled to the linear address generator, adapted to generate a first physical address based on processing all of said calculated linear address; and

c) a first memory storing partial physical address information associated with said first virtual address;

d) a comparator which determines if a second physical address created by a fast address translation of a second virtual address is valid by checking said partial physical address information against a corresponding portion of a complete translation of said second physical address.

14. The system of claim 13, wherein said partial physical address information and said corresponding portion are page frames.

15. The system of claim 13, wherein said second physical address is used for a memory access if said fast address translation is not valid.

* * * * *

EXHIBIT F

US005838986A

United States Patent [19]

Garg et al.

[11] **Patent Number: 5,838,986**

[45] **Date of Patent: *Nov. 17, 1998**

[54] **RISC MICROPROCESSOR ARCHITECTURE IMPLEMENTING MULTIPLE TYPED REGISTER SETS**

[75] Inventors: **Sanjiv Garg**, Fremont; **Derek J. Lentz**, Los Gatos; **Le Trong Nguyen**, Monte Sereno; **Sho Long Chen**, Saratoga, all of Calif.

[73] Assignee: **Seiko Epson Corporation**, Tokyo, Japan

[*] Notice: The term of this patent shall not extend beyond the expiration date of Pat. No. 5,493,687.

[21] Appl. No.: **937,361**

[22] Filed: **Sep. 25, 1997**

Related U.S. Application Data

[63] Continuation of Ser. No. 665,845, Jun. 19, 1996, Pat. No. 5,682,546, which is a continuation of Ser. No. 465,239, Jun. 5, 1995, Pat. No. 5,560,035, which is a continuation of Ser. No. 726,773, Jul. 8, 1991, Pat. No. 5,493,687.

[51] **Int. Cl.**^{6} **G06F 9/34**

[52] **U.S. Cl.** **395/800.23**; 395/569

[58] **Field of Search** 395/800.23, 800.24, 395/309, 310, 390, 391, 393, 569; 711/149

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,212,076	7/1980	Conners .	
5,125,092	6/1992	Prener .	
5,201,056	4/1993	Daniel et al. .	
5,241,636	8/1993	Kohn .	
5,487,156	1/1996	Popescu et al. .	
5,493,687	2/1996	Garg et al.	395/800.23
5,560,035	9/1996	Garg et al.	395/800.23
5,682,546	10/1997	Garg et al.	395/800.23

FOREIGN PATENT DOCUMENTS

0170284	2/1986	European Pat. Off. .
0213843	3/1987	European Pat. Off. .
0241909	10/1987	European Pat. Off. .
0454636	10/1991	European Pat. Off. .
2190521	11/1987	United Kingdom .

OTHER PUBLICATIONS

Patterson et al., "A VLSI RISC," *IEEE Computor,* vol. 15, No. 9, pp. 8–18, Sep. 1982.

Maejima et al., "A 16-bit Microprocessor with Multi-Register Bank Architecture", *Proc. Fall Joint Computor Conference,* Nov. 2–6, 1986, pp. 1014–1019.

Birman et al., "Design of a High-Speed Arithmetic Datapath," *IEEE,* pp. 214–216, 1988.

Ruby B. Lee, "Precision Architecture," *IEEE Computer,* pp. 78–91, Jan. 1989.

(List continued on next page.)

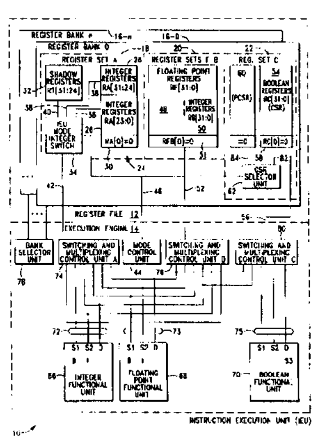
Primary Examiner—Larry D. Donaghue

Attorney, Agent, or Firm—Sterne, Kessler, Goldstein & Fox P.L.L.C.

[57] **ABSTRACT**

A register system for a data processor which operates in a plurality of modes. The register system provides multiple, identical banks of register sets, the data processor controlling access such that instructions and processes need not specify any given bank. An integer register set includes first (RA[23:0]) and second (RA[31:24]) subsets, and a shadow subset (RT[31:24]). While the data processor is in a first mode, instructions access the first and second subsets. While the data processor is in a second mode, instructions may access the first subset, but any attempts to access the second subset are re-routed to the shadow subset instead, transparently to the instructions, allowing system routines to seemingly use the second subset without having to save and restore data which user routines have written to the second subset. A re-typable register set provides integer width data and floating point width data in response to integer instructions and floating point instructions, respectively. Boolean comparison instructions specify particular integer or floating point registers for source data to be compared, and specify a particular Boolean register for the result, so there are no dedicated, fixed-location status flags. Boolean combinational instructions combine specified Boolean registers, for performing complex Boolean comparisons without intervening conditional branch instructions, to minimize pipeline disruption.

30 Claims, 9 Drawing Sheets



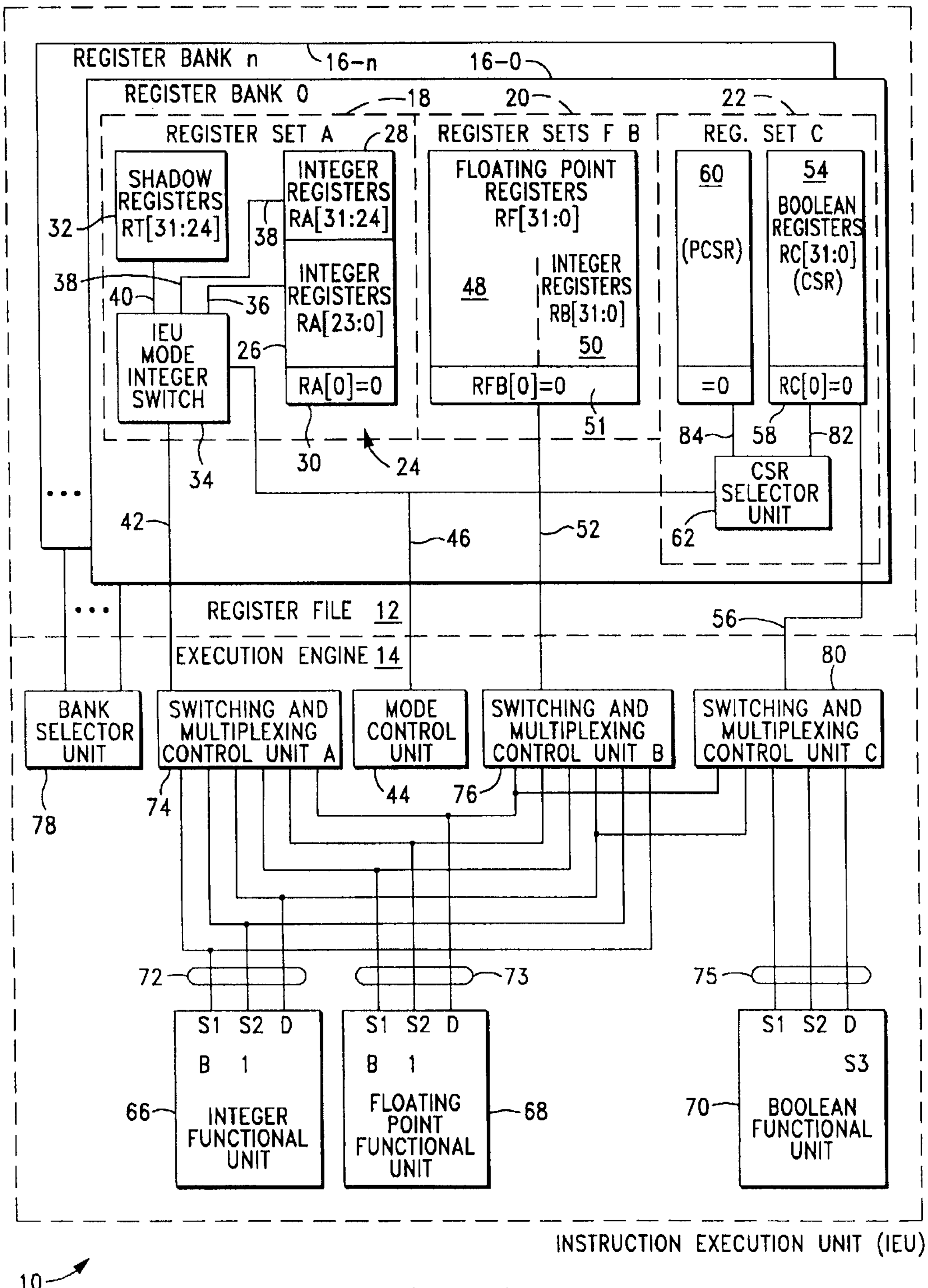
OTHER PUBLICATIONS

Molnar et al., "Floating–Point Processors," *IEEE Intl. Solid–State Circuits Conf.*, pp. 48–49, plus Figure 1, Feb. 1989.
Steven et al., "Harp: A Parallel Pipelined RISC Processor," *Microprocessors and Microsystems,* vol. 13, No. 9, pp. 579–586, Nov. 1989.
Groves et al., "An IBM Second Generation RISC Processor Architecture", *35TH IEEE Computer Society International Conference,* Feb. 26, 1990, pp. 166–172.
Miller et al., "Exploiting Large Register Sets", *Microprocessors and Microsystems,* vol. 14, No. 6, Jul. 1990, pp. 333–340.

Adams et al., "Utilising Low Level Parallelism in General Purpose Code: The HARP Project", *Microprocessing and Micrprogramming,* vol. 29, No. 3, Oct. 1990, pp. 137–149.

Daryl Odnert et al., "Architecture and Computer Enhancements for PA–RISC Workstations," *Proc. from IEEE Compcon,* San Francisco, CA, pp. 214–218, Feb. 1991.

Colin Hunter, "*Series 3200 Programmer's Reference Manual,* " Prentice–Hall Inc., Englewood Cliffs, NJ, 1987, pp. 2–4, 2–21, 2–23, 6–14, and 6–126.



REGISTER BANK n
16-n
16-0
REGISTER BANK 0
18
20
22
REGISTER SET A
28
REGISTER SETS F B
REG. SET C
SHADOW REGISTERS RT[31:24]
INTEGER REGISTERS RA[31:24]
FLOATING POINT REGISTERS RF[31:0]
60
(PCSR)
54
BOOLEAN REGISTERS RC[31:0] (CSR)
32
38
38
40
36
INTEGER REGISTERS RA[23:0]
48
INTEGER REGISTERS RB[31:0]
50
IEU MODE INTEGER SWITCH
26
RA[0]=0
RFB[0]=0
=0
RC[0]=0
51
84
58
82
30
24
34
CSR SELECTOR UNIT
62
42
46
52
REGISTER FILE 12
56
EXECUTION ENGINE 14
80
BANK SELECTOR UNIT
SWITCHING AND MULTIPLEXING CONTROL UNIT A
MODE CONTROL UNIT
SWITCHING AND MULTIPLEXING CONTROL UNIT B
SWITCHING AND MULTIPLEXING CONTROL UNIT C
78
74
44
76
72
73
75
S1 S2 D
B 1
INTEGER FUNCTIONAL UNIT
66
S1 S2 D
B 1
FLOATING POINT FUNCTIONAL UNIT
68
S1 S2 D
S3
BOOLEAN FUNCTIONAL UNIT
70
INSTRUCTION EXECUTION UNIT (IEU)
10

FIG.—1

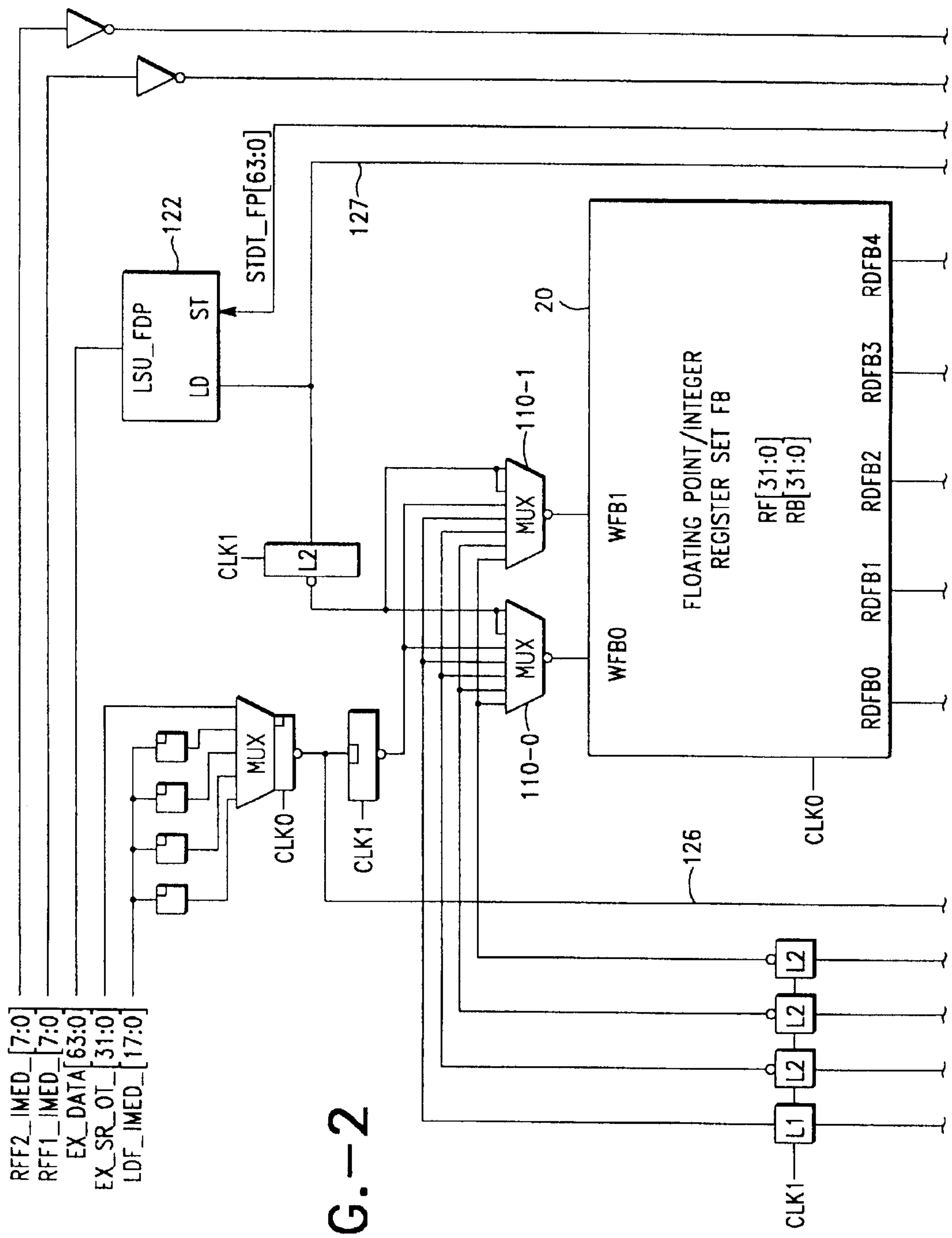

RFF2_IMED_[7:0]
RFF1_IMED_[7:0]
EX_DATA[63:0]
EX_SR_OT_[31:0]
LDF_IMED_[17:0]
LSU_FDP
LD
ST
122
STDT_FP[63:0]
127
MUX
CLK0
CLK1
L2
110-1
110-0
WFB1
WFB0
20
FLOATING POINT/INTEGER
REGISTER SET FB
RF[31:0]
RB[31:0]
RDFB4
RDFB3
RDFB2
RDFB1
RDFB0
126
L1

FIG.—2

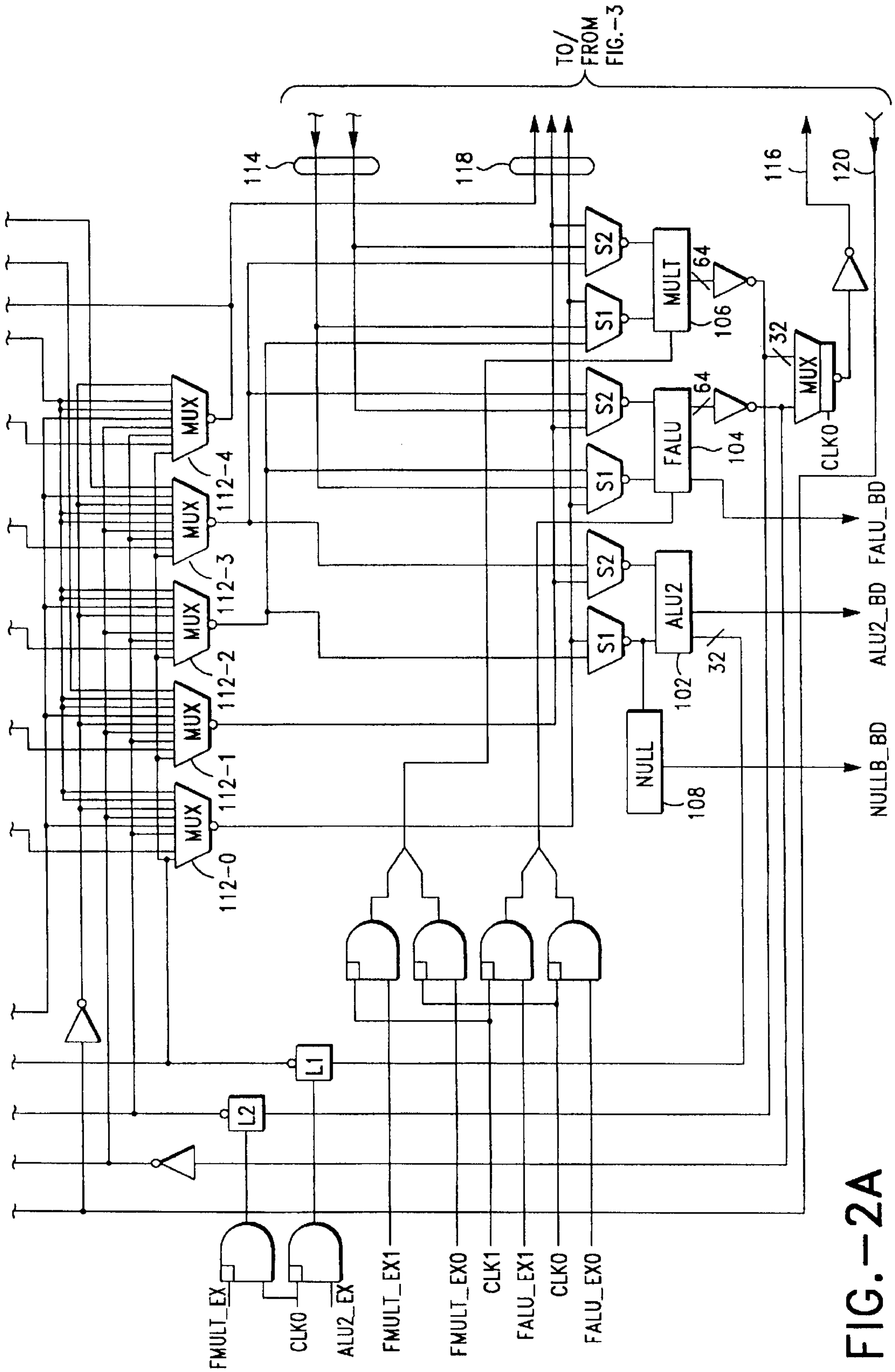

TO/
FROM
FIG.—3
114
118
116
120
S2
S1
MULT
64
106
32
MUX
FALU
104
CLK0
ALU2
102
32
NULL
108
MUX
112-4
112-3
112-2
112-1
112-0
L1
L2
FALU_BD
ALU2_BD
NULLB_BD
FMULT_EX
CLK0
ALU2_EX
FMULT_EX1
FMULT_EX0
CLK1
FALU_EX1
CLK0
FALU_EX0

FIG.—2A

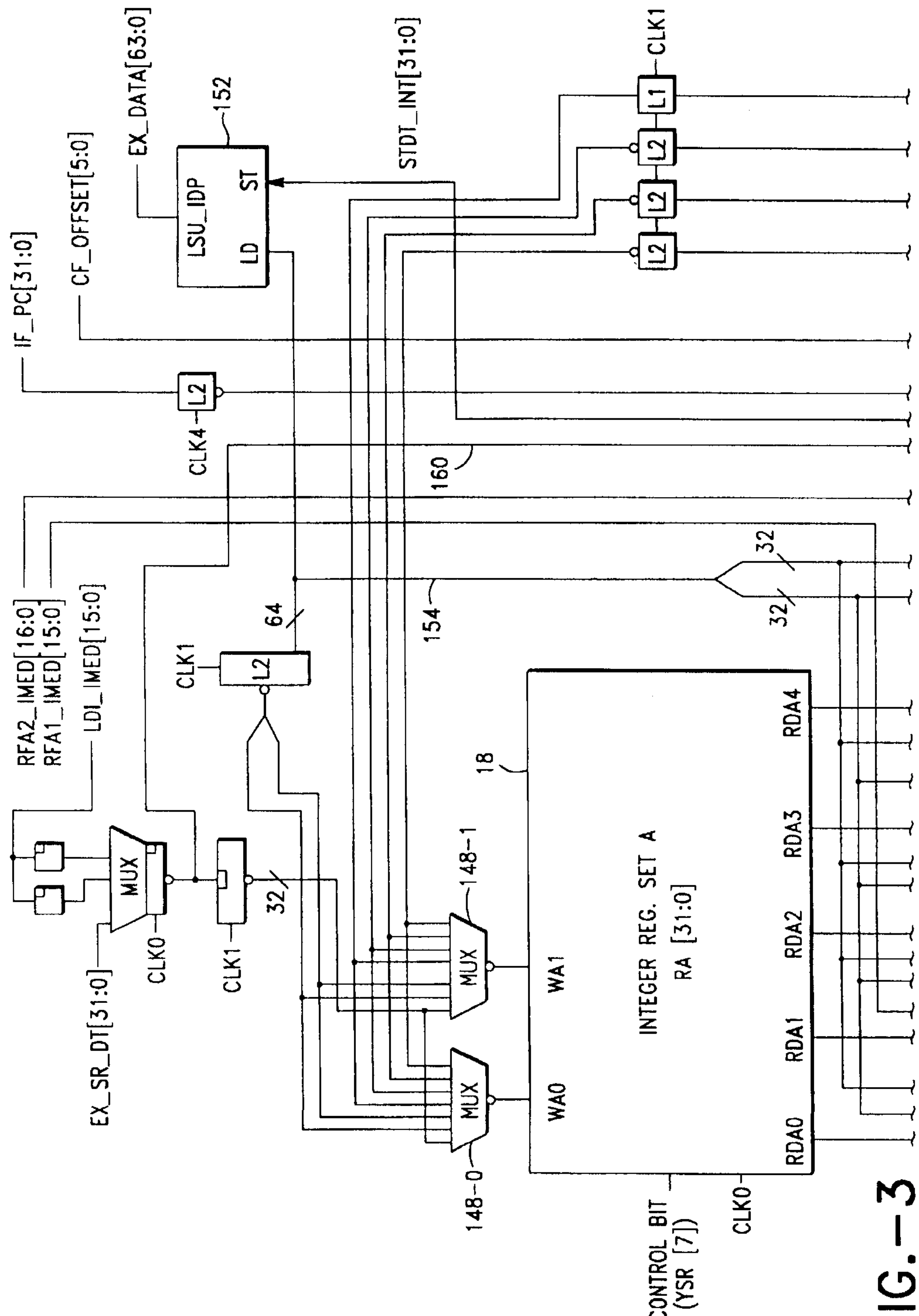

IF_PC[31:0]
CF_OFFSET[5:0]
EX_DATA[63:0]
152
LSU_IDP
LD
ST
STDT_INT[31:0]
CLK1
L1
L2
L2
L2
L2
CLK4
160
RFA2_IMED[16:0]
RFA1_IMED[15:0]
LDI_IMED[15:0]
64
L2
CLK1
154
32
32
MUX
CLK0
CLK1
32
EX_SR_DT[31:0]
148-1
MUX
MUX
148-0
18
WA1
WA0
INTEGER REG. SET A
RA [31:0]
RDA4
RDA3
RDA2
RDA1
RDA0
CONTROL BIT
(YSR [7])
CLK0

FIG.—3

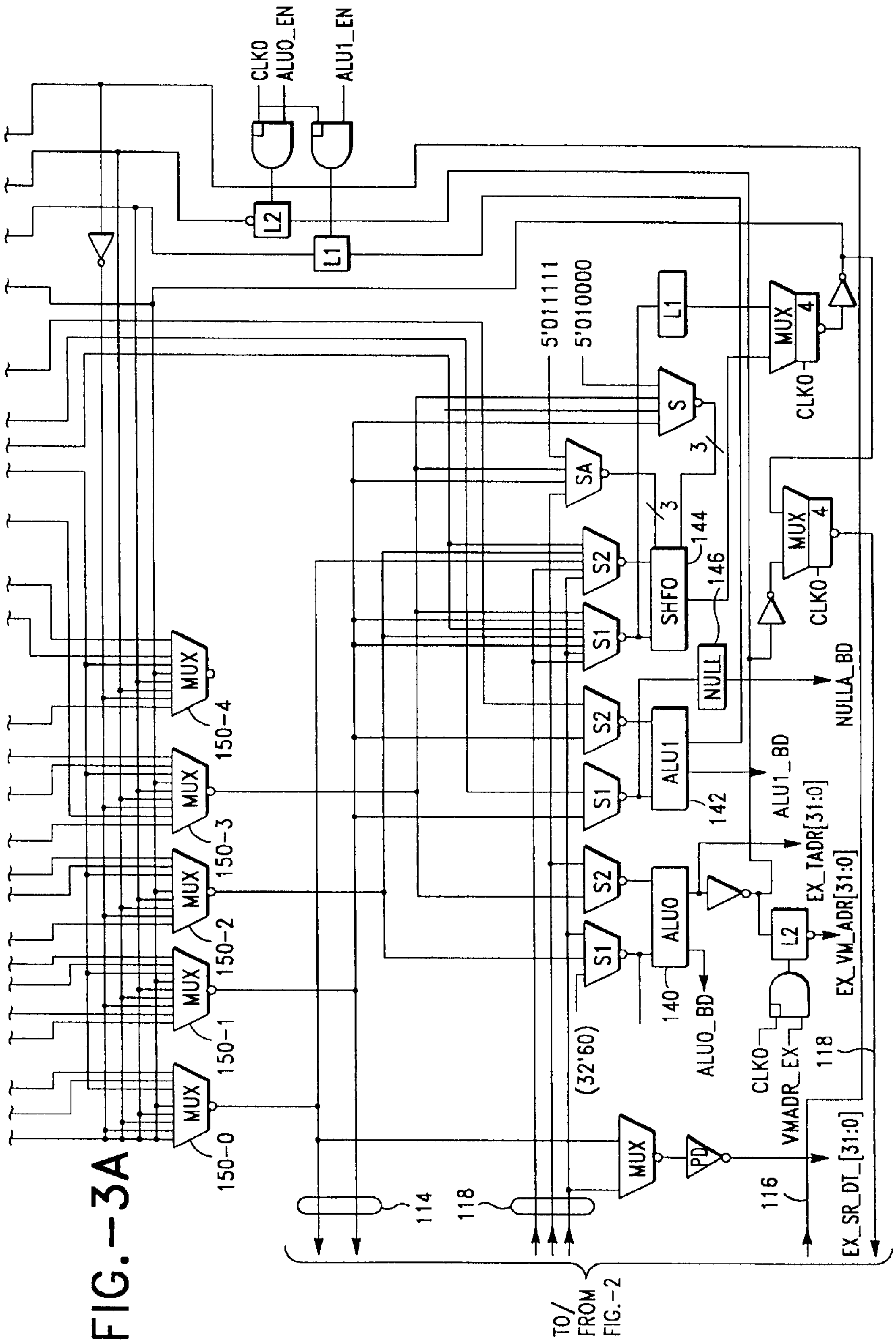

CLK0
ALU0_EN
ALU1_EN
L2
L1
5'011111
5'010000
MUX
4
CLK0
S
3
SA
3
144
146
S2
S1
SHF0
NULL
NULLA_BD
S2
S1
ALU1
142
ALU1_BD
EX_TADR[31:0]
S2
S1
ALU0
140
ALU0_BD
L2
EX_VM_ADR[31:0]
CLK0
VMADR_EX
118
(32'60)
MUX
150-0
150-1
150-2
150-3
150-4
MUX
PD
116
EX_SR_DT_[31:0]
114
118
TO/
FROM
FIG.–2

FIG.—3A

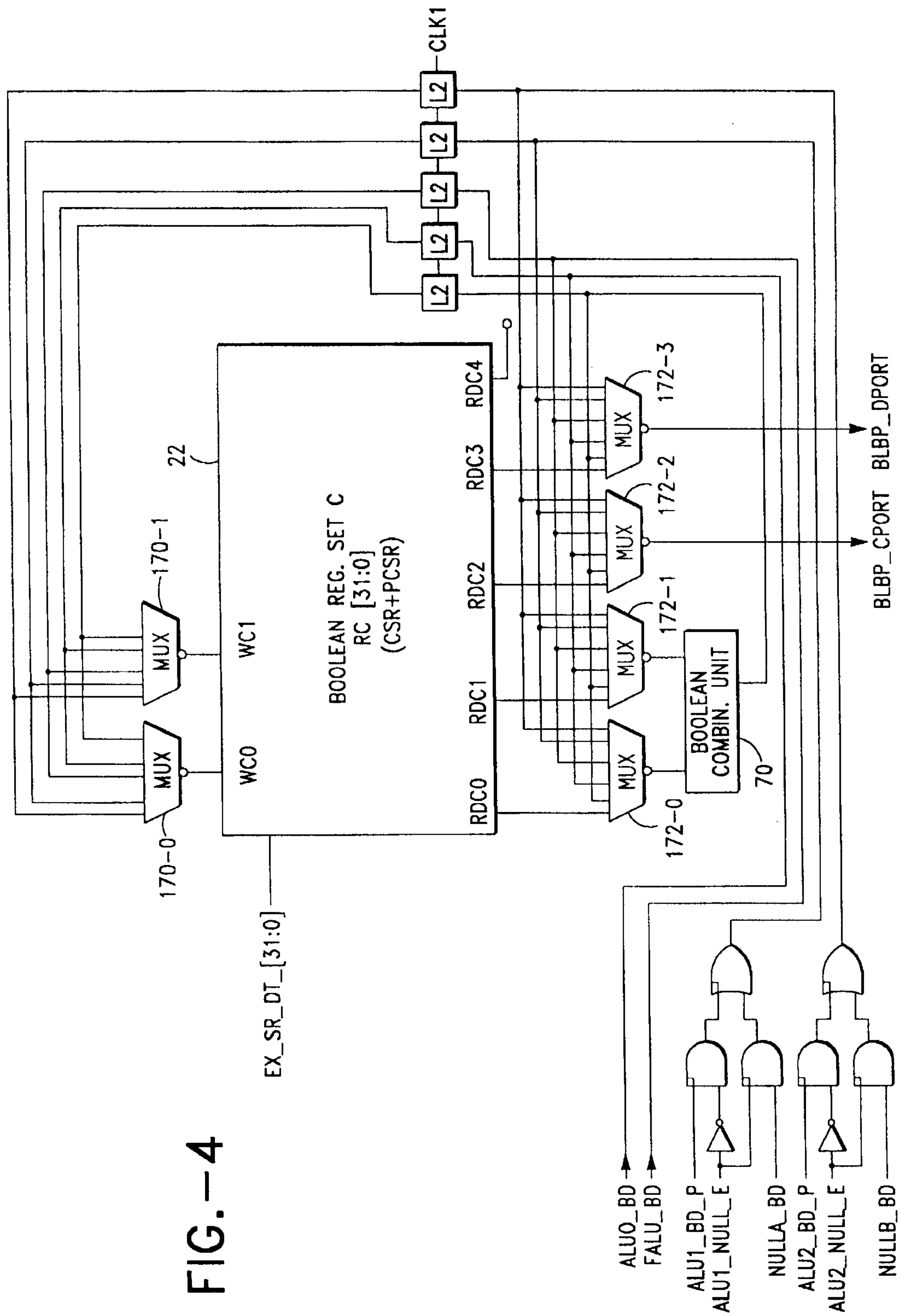
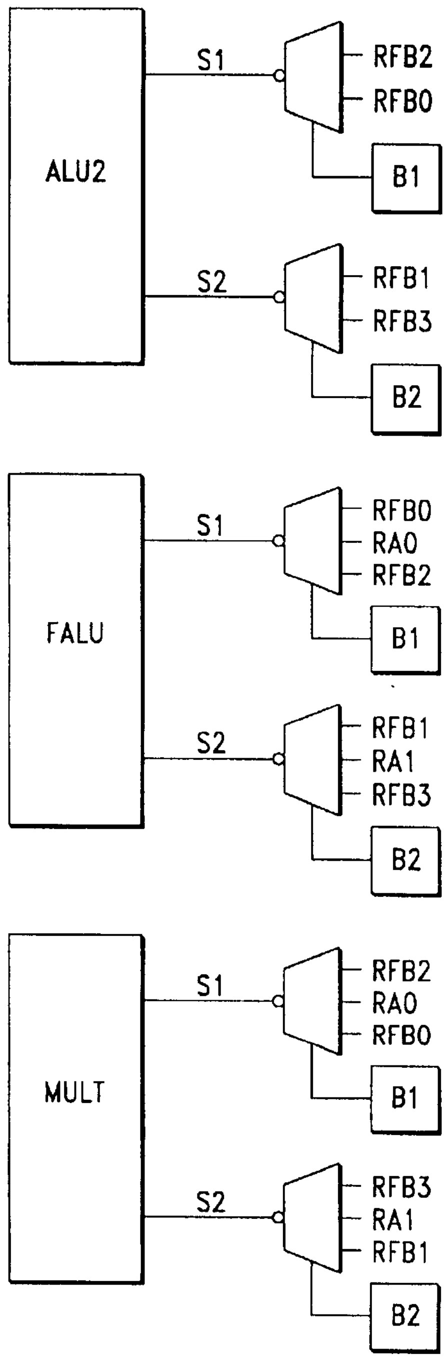

CLK1
L2
L2
L2
L2
L2
22
BOOLEAN REG. SET C
RC [31:0]
(CSR+PCSR)
170-1
170-0
MUX
MUX
WC1
WC0
RDC4
RDC3
RDC2
RDC1
RDC0
172-3
172-2
172-1
172-0
MUX
MUX
MUX
MUX
BOOLEAN
COMBIN. UNIT
70
BLBP_DPORT
BLBP_CPORT
EX_SR_DT_[31:0]
ALU0_BD
FALU_BD
ALU1_BD_P
ALU1_NULL_E
NULLA_BD
ALU2_BD_P
ALU2_NULL_E
NULLB_BD

FIG.—4



ALU2
S1
RFB2
RFB0
B1
S2
RFB1
RFB3
B2
FALU
S1
RFB0
RA0
RFB2
B1
S2
RFB1
RA1
RFB3
B2
MULT
S1
RFB2
RA0
RFB0
B1
S2
RFB3
RA1
RFB1
B2

FIG.—5

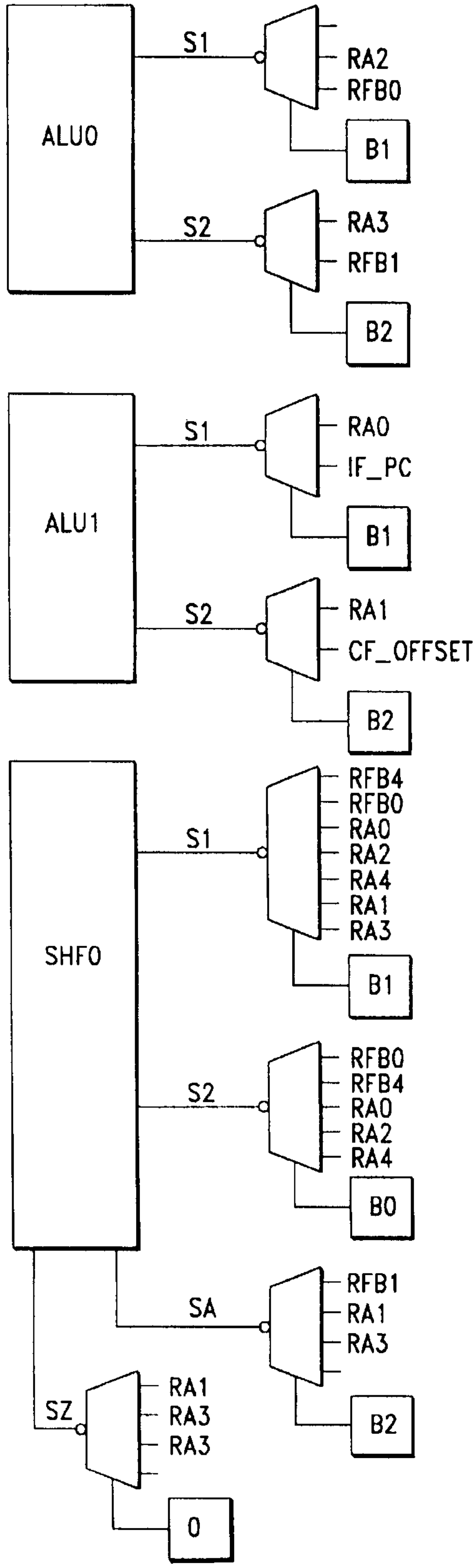
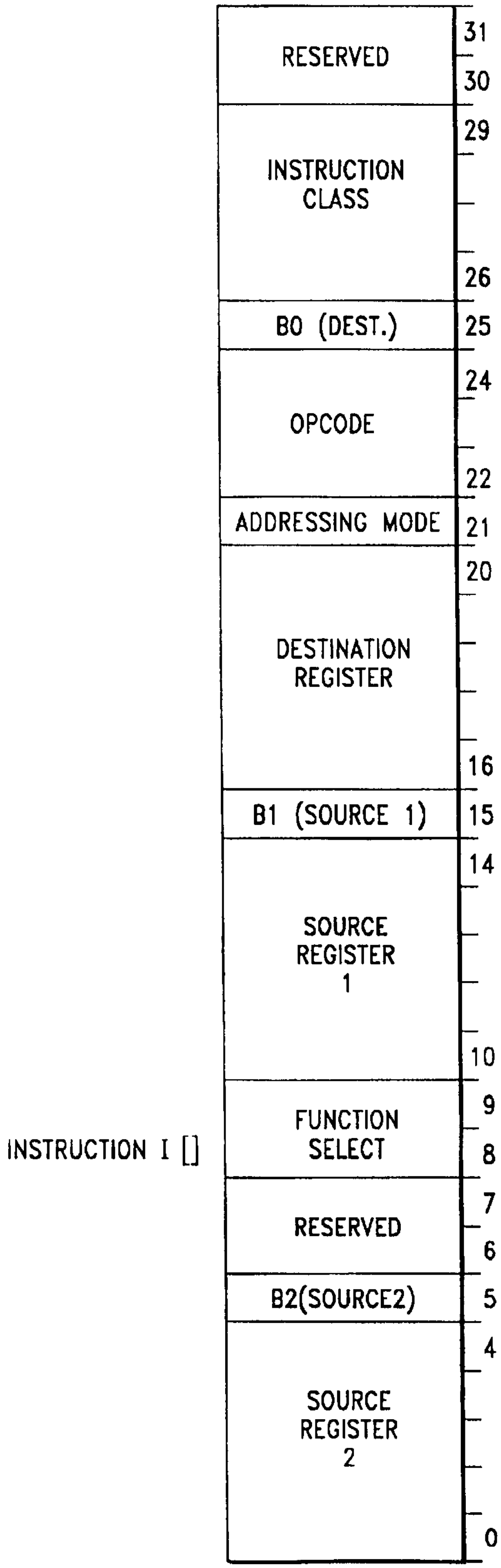

ALU0
S1
RA2
RFB0
B1
S2
RA3
RFB1
B2
ALU1
S1
RA0
IF_PC
B1
S2
RA1
CF_OFFSET
B2
SHF0
S1
RFB4
RFB0
RA0
RA2
RA4
RA1
RA3
B1
S2
RFB0
RFB4
RA0
RA2
RA4
B0
SA
RFB1
RA1
RA3
B2
SZ
RA1
RA3
RA3
0

FIG.—6



RESERVED
31
30
INSTRUCTION CLASS
29
26
B0 (DEST.)
25
OPCODE
24
22
ADDRESSING MODE
21
DESTINATION REGISTER
20
16
B1 (SOURCE 1)
15
SOURCE REGISTER 1
14
10
FUNCTION SELECT
9
8
RESERVED
7
6
B2(SOURCE2)
5
SOURCE REGISTER 2
4
0
INSTRUCTION I []

FIG.—7

RISC MICROPROCESSOR ARCHITECTURE IMPLEMENTING MULTIPLE TYPED REGISTER SETS

This application is a continuation of application Ser. No. 08/665,845, filed on Jun. 19, 1996 (allowed), which is a continuation of application Ser. No. 08/465,239, filed Jun. 5, 1995, (patented) now U.S. Pat. No. 5,560,035 which is a continuation of application Ser. No. 07/726,773, filed Jul. 8, 1991, (patented), now U.S. Pat. No. 5,493,687

CROSS-REFERENCE TO RELATED APPLICATIONS

Applications of particular interest to the present application, include:

1. High-Performance, Superscalar-Based Computer System with Out-of-Order Instruction Execution, application Ser. No. 07/817,810, filed Jan. 8, 1992, now U.S. Pat. No. 5,539,911, by Le Trong Nguyen et al.;

2. High-Performance Superscalar-Based Computer System with Out-of-Order Instruction Execution and Concurrent Results Distribution, application Ser. No. 08/397,016, filed Mar. 1, 1995, now U.S. Pat. No. 5,560,032, by Quang Trang et al.;

3. RISC Microprocessor Architecture with Isolated Architectural Dependencies, application Ser. No. 08/292,177, filed Aug. 18, 1994, now abandoned, which is an FWC of application Ser. No. 07/817,807, filed Jan. 8, 1992, which is a continuation of application Ser. No. 07/726,744, filed Jul. 8, 1991, by Yoshiyuki Miyayama;

4. RISC Microprocessor Architecture Implementing Fast Trap and Exception State, application Ser. No. 08/345,333, filed Nov. 21, 1994, now U.S. Pat. No. 5,481,685, by Quang Trang;

5. Page Printer Controller Including a Single Chip Superscalar Microprocessor with Graphics Functional Units, application Ser. No. 08/267,646, filed Jun. 28, 1994, now U.S. Pat. No. 5,394,515, by Derek Lentz et al.; and

6. Microprocessor Architecture Capable with a Switch Network for Data Transfer Between Cache, Memory Port, and IOU, application Ser. No. 07/726,893, filed Jul. 8, 1991, now U.S. Pat. No. 5,440,752, by Derek Lentz et al.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates generally to microprocessors, and more specifically to a RISC microprocessor having plural, symmetrical sets of registers.

2. Description of the Background

In addition to the usual complement of main memory storage and secondary permanent storage, a microprocessor-based computer system typically also includes one or more general purpose data registers, one or more address registers, and one or more status flags. Previous systems have included integer registers for holding integer data and floating point registers for holding floating point data. Typically, the status flags are used for indicating certain conditions resulting from the most recently executed operation. There generally are status flags for indicating whether, in the previous operation: a carry occurred, a negative number resulted, and/or a zero resulted.

These flags prove useful in determining the outcome of conditional branching within the flow of program control. For example, if it is desired to compare a first number to a

second number and upon the conditions that the two are equal, to branch to a given subroutine, the microprocessor may compare the two numbers by subtracting one from the other, and setting or clearing the appropriate condition flags. The numerical value of the result of the subtraction need not be stored. A conditional branch instruction may then be executed, conditioned upon the status of the zero flag. While being simple to implement, this scheme lacks flexibility and power. Once the comparison has been performed, no further numerical or other operations may be performed before the conditional branch upon the appropriate flag; otherwise, the intervening instructions will overwrite the condition flag values resulting from the comparison, likely causing erroneous branching. The scheme is further complicated by the fact that it may be desirable to form greatly complex tests for branching, rather than the simple equality example given above.

For example, assume that the program should branch to the subroutine only upon the condition that a first number is greater than a second number, and a third number is less than a fourth number, and a fifth number is equal to a sixth number. It would be necessary for previous microprocessors to perform a lengthy series of comparisons heavily interspersed with conditional branches. A particularly undesirable feature of this serial scheme of comparing and branching is observed in any microprocessor having an instruction pipeline.

In a pipelined microprocessor, more than one instruction is being executed at any given time, with the plural instructions being in different stages of execution at any given moment. This provides for vastly improved throughput. A typical pipeline microprocessor may include pipeline stages for: (a) fetching an instruction, (b) decoding the instruction, (c) obtaining the instruction's operands, (d) executing the instruction, and (e) storing the results. The problem arises when a conditional branch instruction is fetched. It may be the case that the conditional branch's condition cannot yet be tested, as the operands may not yet be calculated, if they are to result from operations which are yet in the pipeline. This results in a "pipeline stall", which dramatically slows down the processor.

Another shortcoming of previous microprocessor-based systems is that they have included only a single set of registers of any given data type. In previous architectures, when an increased number of registers has been desired within a given data type, the solution has been simply to increase the size of the single set of those type of registers. This may result in addressing problems, access conflict problems, and symmetry problems.

On a similar note, previous architectures have restricted each given register set to one respective numerical, data type. Various prior systems have allowed general purpose registers to hold either numerical data or address "data", but the present application will not use the term "data" to include addresses. What is intended may be best understood with reference to two prior systems. The Intel 8085 microprocessor includes a register pair "HL" which can be used to hold either two bytes of numerical data or one two-byte address. The present application's improvement is not directed to that issue. More on point, the Intel 80486 microprocessor includes a set of general purpose integer data registers and a set of floating point registers, with each set being limited to its respective data type, at least for purposes of direct register usage by arithmetic and logic units.

This proves wasteful of the microprocessor's resources, such as the available silicon area, when the microprocessor

is performing operations which do not involve both data types. For example, user applications frequently involve exclusively integer operations, and perform no floating point operations whatsoever. When such a user application is run on a previous microprocessor which includes floating point registers (such as the 80486), those floating point registers remain idle during the entire execution.

Another problem with previous microprocessor register set architecture is observed in context switching or state switching between a user application and a higher access privilege level entity such as the operating system kernel. When control within the microprocessor switches context, mode, or state, the operating system kernel or other entity to which control is passed typically does not operate on the same data which the user application has been operating on. Thus, the data registers typically hold data values which are not useful to the new control entity but which must be maintained until the user application is resumed. The kernel must generally have registers for its own use, but typically has no way of knowing which registers are presently in use by the user application. In order to make space for its own data, the kernel must swap out or otherwise store the contents of a predetermined subset of the registers. This results in considerable loss of processing time to overhead, especially if the kernel makes repeated, short-duration assertions of control.

On a related note, in prior microprocessors, when it is required that a "grand scale" s context switch be made, it has been necessary for the microprocessor to expend even greater amounts of processing resources, including a generally large number of processing cycles, to save all data and state information before making the switch. When context is switched back, the same performance penalty has previously been paid, to restore the system to its former state. For example, if a microprocessor is executing two user applications, each of which requires the full complement of registers of each data type, and each of which may be in various stages of condition code setting operations or numerical calculations, each switch from one user application to the other necessarily involves swapping or otherwise saving the contents of every data register and state flag in the system. This obviously involves a great deal of operational overhead, resulting in significant performance degradation, particularly if the main or the secondary storage to which the registers must be saved is significantly slower than the microprocessor itself.

Therefore, we have discovered that it is desirable to have an improved microprocessor architecture which allows the various component conditions of a complex condition to be calculated without any intervening conditional branches. We have further discovered that it is desirable that the plural simple conditions be calculable in parallel, to improve throughput of the microprocessor.

We have also discovered that it is desirable to have an architecture which allows multiple register sets within a given data type.

Additionally, we have discovered it to be desirable for a microprocessor's floating point registers to be usable as integer registers, in case the available integer registers are inadequate to optimally to hold the necessary amount of integer data. Notably, we have discovered that it is desirable that such re-typing be completely transparent to the user application.

We have discovered it to be highly desirable to have a microprocessor which provides a dedicated subset of registers which are reserved for use by the kernel in lieu of at least a subset of the user registers, and that this new set of registers should be addressable in exactly the same manner as the register subset which they replace, in order that the kernel may use the same register addressing scheme as user applications. We have further observed that it is desirable that the switch between the two subsets of registers require no microprocessor overhead cycles, in order to maximally utilize the microprocessor's resources.

Also, we have discovered it to be desirable to have a microprocessor architecture which allows for a "grand scale" context switch to be performed with minimal overhead. In this vein, we have discovered that is desirable to have an architecture which allows for plural banks of register sets of each type, such that two or more user applications may be operating in a multi-tasking environment, or other "simultaneous" mode, with each user application having sole access to at least a full bank of registers. It is our discovery that the register addressing scheme should, desirably, not differ between user applications, nor between register banks, to maximize simplicity of the user applications, and that the system should provide hardware support for switching between the register banks so that the user applications need not be aware of which register bank which they are presently using or even of the existence of other register banks or of other user applications.

These and other advantages of our invention will be appreciated with reference to the following description of our invention, the accompanying drawings, and the claims.

SUMMARY OF THE INVENTION

The present invention provides a register file system comprising: an integer register set including first and second subsets of Integer registers, and a shadow subset; a re-typable set of registers which are individually usable as integer registers or as floating point registers; and a set of individually addressable Boolean registers.

The present invention includes integer and floating point functional units which execute integer instructions accessing the integer register set, and which operate in a plurality of modes. In any mode, instructions are granted ordinary access to the first subset of integer registers. In a first mode, instructions are also granted ordinary access to the second subset. However, in a second mode, instructions attempting to access the second subset are instead granted access to the shadow subset, in a manner which is transparent to the instructions. Thus, routines may be written without regard to which mode they will operate in, and system routines (which operate in the second mode) can have at least the second subset seemingly at their disposal, without having to expend the otherwise-required overhead of saving the second subset's contents (which may be in use by user processes operating in the first mode).

The invention further includes a plurality of integer register sets, which are individually addressable as specified by fields in instructions. The register sets include read ports and write ports which are accessed by multiplexers, wherein the multiplexers are controlled by contents of the register set-specifying fields in the instructions.

One of the integer register sets is also usable as a floating point register set. In one embodiment, this set is sixty-four bits wide to hold double-precision floating point data, but only the low order thirty-two bits are used by integer instructions.

The invention includes functional units for performing Boolean operations, and further includes a Boolean register

set for holding results of the Boolean operations such that no dedicated, fixed-location status flags are required. The integer and floating point functional units execute numerical comparison instructions, which specify individual ones of the Boolean registers to hold results of the comparisons. A Boolean functional unit executes Boolean combinational instructions whose sources and destination are specified registers in the Boolean register set. Thus, the present invention may perform conditional branches upon a single result of a complex Boolean function without intervening conditional branch instructions between the fundamental parts of the complex Boolean function, minimizing pipeline disruption in the data processor.

Finally, there are multiple, identical register banks in the system, each bank including the above-described register sets. A bank may be allocated to a given process or routine, such that the instructions within the routine need not specify upon which bank they operate.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a block diagram of the instruction execution unit of the microprocessor of the present invention, showing the elements of the register file.

FIGS. **2**,**2***a*, **3**,**3***a* and **4** are simplified schematic and block diagrams of the floating point, integer and Boolean portions of the instruction execution unit of FIG. **1**, respectively.

FIGS. **5**–**6** are more detailed views of the floating point and integer portions, respectively, showing the means for selecting between register sets.

FIG. **7** illustrates the fields of an exemplary microprocessor instruction word executable by the instruction execution unit of FIG. **1**.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

I. REGISTER FILE

FIG. **1** illustrates the basic components of the instruction execution unit (IEU) **10** of the RISC (reduced instruction set computing) processor of the present invention. The IEU **10** includes a register file **12** and an execution engine **14**. The register file **12** includes one or more register banks **16-0** to **16**-n. It will be understood that the structure of each register bank **16** is identical to all of the other register banks **16**. Therefore, the present application will describe only register bank **16-0**. The register bank includes a register set A **18**, a register set FB **20**, and a register set C **22**.

In general, the invention may be characterized as a RISC microprocessor having a register file optimally configured for use in the execution of RISC instructions, as opposed to conventional register files-which are sufficient for use in the execution of CISC (complex instruction set computing) instructions by CISC processors. By having a specially adapted register file, the execution engine of the microprocessor's IEU achieves greatly improved performance, both in terms of resource utilization and in terms of raw throughput. The general concept is to tune a register set to a RISC instruction, while the specific implementation may involve any of the register sets in the architecture.

A. Register Set A

Register set A **18** includes integer registers **24** (RA[31:0]), each of which is adapted to hold an integer value datum. In one embodiment, each integer may be thirty-two bits wide. The RA[] integer registers **24** include a first plurality **26** of integer registers (RA[23:0]) and a second plurality **28** of integer registers (RA[31:24]). The RA[] integer registers **24** are each of identical structure, and are each addressable in the same manner, albeit with a unique address within the integer register set **24**. For example, a first integer register **30** (RA[0]) is addressable at a zero offset within the integer register set **24**.

RA[0] always contains the value zero. It has been observed that user applications and other programs use the constant value zero more than any other constant value. It is, therefore, desirable to have a zero readily available at all times, for clearing, comparing, and other purposes. Another advantage of having a constant, hard-wired value in a given register, regardless of the particular value, is that the given register may be used as the destination of any instruction whose results need not be saved.

Also, this means that the fixed register will never be the cause of a data dependency delay. A data dependency exists when a "slave" instruction requires, for one or more of its operands, the result of a "master" instruction. In a pipelined processor, this may cause pipeline stalls. For example, the master instruction, although occurring earlier in the code sequence than the slave instruction, may take considerably longer to execute. It will be readily appreciated that if a slave "increment and store" instruction operates on the result data of a master wquadruple-word integer divide instruction, the slave instruction will be fetched, decoded, and awaiting execution many clock cycles before the master instruction has finished execution. However, in certain instances, the numerical result of a master instruction is not needed, and the master instruction is executed for some other purpose only, such as to set condition code flags. If the master instruction's destination is RA[0], the numerical results will be effectively discarded. The data dependency checker (not shown) of the IEU **10** will not cause the slave instruction to be delayed, as the ultimate result of the master instruction—zero—is already known.

The integer register set A **24** also includes a set of shadow registers **32** (RT[31:24]). Each shadow register can hold an integer value, and is, in one embodiment, also thirty-two bits wide. Each shadow register is addressable as an offset in the same manner in which each integer register is addressable.

Finally, the register set A includes an IEU mode integer switch **34**. The switch **34**, like other such elements, need not have a physical embodiment as a switch, so long as the corresponding logical functionality is provided within the register sets. The IEU mode integer switch **34** is coupled to the first subset **26** of integer registers on line **36**, to the second subset of integer registers **28** on line **38**, and to the shadow registers **32** on line **40**. All accesses to the register set A **18** are made through the IEU mode integer switch **34** on line **42**. Any access request to read or write a register in the first subset RA[23:0] is passed automatically through the IEU mode integer switch **34**. However, accesses to an integer register with an offset outside the first subset RA[23:0] will be directed either to the second subset RA[31:24] or the shadow registers RT[31;24], depending upon the operational mode of the execution engine **14**.

The IEU mode integer switch **34** is responsive to a mode control unit **44** in the execution engine **14**. The mode control unit **44** provides pertinent state or mode information about the IEU **10** to the IEU mode integer switch **34** on line **46**. When the execution engine performs a context switch such as a transfer to kernel mode, the mode control unit **44** controls the IEU mode integer switch **34** such that any requests to the second subset RA[31:24] are re-directed to the shadow RT[31:24], using the same requested offset within the integer set. Any operating system kernel or other then-executing entity may thus have apparent access to the

second subset RA[31:24] without the otherwise-required overhead of swapping the contents of the second subset RA[31:24] out to main memory, or pushing the second subset RA[31:24] onto a stack, or other conventional register-saving technique.

When the execution engine **14** returns to normal user mode and control passes to the originally-executing user application, the mode control unit **44** controls the IEU mode integer switch **34** such that access is again directed to the second subset RA[31:24]. In one embodiment, the mode control unit **44** is responsive to the present state of interrupt enablement in the IEU **10**. In one embodiment, the execution engine **14** includes a processor status register (PSR) (not shown), which includes a one-bit flag (PSR[7]) indicating whether interrupts are enabled or disabled. Thus, the line **46** may simply couple the IEU mode integer switch **34** to the interrupts-enabled flag in the PSR. While interrupts are disabled, the IEU **10** maintains access to the integers RA[23:0], in order that it may readily perform analysis of various data of the user application. This may allow improved debugging, error reporting, or system performance analysis.

B. Register Set FB

The re-typable register set FB **20** may be thought of as including floating point registers **48** (RF[31:0]); and/or integer registers **50** (RB[31:0]). When neither data type is implied to the exclusion of the other, this application will use the term RFB[]. In one embodiment, the floating point registers RF[] occupy the same physical silicon space as the integer registers RB[]. In one embodiment, the floating point registers RF[] are sixty-four bits wide and the integer registers RB[] are thirty-two bits wide. It will be understood that if double-precision floating point numbers are not required, the register set RFB[] may advantageously be constructed in a thirty-two-bit width to save the silicon area otherwise required by the extra thirty-two bits of each floating point register.

Each individual register in the register set RFB[] may hold either a floating point value or an integer value. The register set RFB[] may include optional hardware for preventing accidental access of a floating point value as though it were an integer value, and vice versa. In one embodiment, however, in the interest of simplifying the register set RFB[], it is simply left to the software designer to ensure that no erroneous usages of individual registers are made. Thus, the execution engine **14** simply makes an access request on line **52**, specifying an offset into the register set RFB[], without specifying whether the register at the given offset is intended to be used as a floating point register or an integer register. Within the execution engine **14**, various entities may use either the full sixty-four bits provided by the register set RFB[], or may use only the low order thirty-two bits, such as in integer operations or single-precision floating point operations.

A first register RFB[0] **51** contains the constant value zero, in a form such that RB[0] is a thirty-two-bit integer zero (0000_{hex}) and RF[0] is a sixty-four-bit floating point zero (00000000_{hex}) This provides the same advantages as described above for RA[0].

C. Register Set C

The register set C **22** includes a plurality of Boolean registers **54** (RC[31:0]). RC[] is also known as the "condition status register" (CSR). The Boolean registers RC[] are each identical in structure and addressing, albeit that each is individually addressable at a unique address or offset within RC[].

In one embodiment, register set C further includes a "previous condition status register" (PCSR) **60**, and the register set C also includes a CSR selector unit **62**, which is responsive to the mode control unit **44** to select alternatively between the CSR **54** and the PCSR **60**. In the one embodiment, the CSR is used when interrupts are enabled, and the PCSR is used when interrupts are disabled. The CSR and PCSP are identical in all other respects. In the one embodiment, when interrupts are set to be disabled, the CSR selector unit **62** pushes the contents of the CSR into the PCSR, overwriting the former contents of the PCSR, and when interrupts are re-enabled, the CSR selector unit **62** pops the contents of the PCSR back into the CSR. In other embodiments it may be desirable to merely alternate access between the CSR and the PCSR, as is done with RA[31:24] and RT[31:24]. In any event, the PCSR is always available as a thirty-two-bit "special register".

None of the Boolean registers is a dedicated condition flag, unlike the Boolean registers in previously known microprocessors. That is, the CSR **54** does not include a dedicated carry flag, nor a dedicated a minus flag, nor a dedicated flag indicating equality of a comparison or a zero subtraction result. Rather, any Boolean register may be the destination of the Boolean result of any Boolean operation. As with the other register sets, a first Boolean register **58** (RC[0]) always contains the value zero, to obtain the advantages explained above for RA[0]. In the preferred embodiment, each Boolean register is one bit wide, indicating one Boolean value.

II. EXECUTION ENGINE

The execution engine **14** includes one or more integer functional units **66**, one or more floating point functional units **68**, and one or more Boolean functional units **70**. The functional units execute instructions as will be explained below. Buses **72**, **73**, and **75** connect the various elements of the IEU **10**, and will each be understood to represent data, address, and control paths.

A. Instruction Format

FIG. **7** illustrates one exemplary format for an integer instruction which the execution engine **14** may execute. It will be understood that not all instructions need to adhere strictly to the illustrated format, and that the data processing system includes an instruction fetcher and decoder (not shown) which are adapted to operate upon varying format instructions. The single example of FIG. **7** is for ease in explanation only. Throughout this Application the identification I[] will be used to identify various bits of the instruction. I[31:30] are reserved for future implementations of the execution engine **14**. I[29:26] identify the Instruction class of the particular instruction.

Table 1 shows the various classes of instructions performed by the present invention.

TABLE 1

Instruction Classes

Class	Instructions
0–3	Integer and floating point register-to-register instructions
4	Immediate constant load
5	Reserved
6	Load
7	Store
8–11	Control Flow
12	Modifier
13	Boolean operations
14	Reserved
15	Atomic (extended)

Instruction classes of particular interest to this Application include the Class 0–3 register-to-register instructions and the Class 13 Boolean operations. While other classes of instructions also operate upon the register file **12**, further discussion of those classes is not believed necessary in order to fully understand the present invention.

I[25] is identified as B0, and indicates whether the destination register is in register set A or register set B. I[24:22] are an opcode which identifies, within the given instruction class, which specific function is to be performed. For example, within the register-to-register classes, an opcode may specify "addition". I[21] identifies the addressing mode which is to be used when performing the instruction—either register source addressing or immediate source addressing. I[20:16] identify the destination register as an offset within the register set indicated by B0. I[15] is identified as B1 and indicates whether the first operand is to be taken from register set A or register set B. I[14:10] identify the register offset It from which the first operand is to be taken. I[9:8] identify a function selection—an extension of the opcode I[24:22]. I[7:6] are reserved. I[1] is identified as B2 and indicates whether a second operand of the instruction is to be taken from register set A or register set B. Finally, I[4:0] identify the register offset from which the second operand is to be taken.

With reference to FIG. **1**, the integer functional unit **66** and floating point functional unit **68** are equipped to perform integer comparison instructions and floating point comparisons, respectively. The instruction format for the comparison instruction is substantially identical to that shown in FIG. **7**, with the caveat that various fields may advantageously be identified by slightly different names. I[20:16] identifies the destination register where the result is to be stored, but the addressing mode field I[21] does not select between register sets A or B. Rather, the addressing mode field indicates whether the second source of the comparison is found in a register or is immediate data. Because the comparison is a Boolean type instruction, the destination register is always found in register set C. All other fields function as shown in FIG. **7**. In performing Boolean operations within the integer and floating point functional units, the opcode and function select fields identify which Boolean condition is to be tested for in comparing the two operands. The integer and the floating point functional units fully support the IEEE standards for numerical comparisons.

The IEU **10** is a load/store machine. This means that when the contents of a register are stored to memory or read from memory, an address calculation must be performed in order to determine which location in memory is to be the source or the destination of the store or load, respectively. When this is the case, the destination register field I[20:16] identifies the register which is the destination or the source of the load or store, respectively. The source register 1 field, I[14:10], identifies a register in either set A or B which contains a base address of the memory location. In one embodiment, the source register 2 field, I[4:0], identifies a register in set A or set B which contains an index or an offset from the base. The load/store address is calculated by adding the index to the base. In another mode, I[7:0] include immediate data which are to be added as an index to the base.

B. Operation of the Instruction Execution Unit and Register Sets

It will be understood by those skilled in the art that the integer functional unit **66**, the floating point functional unit **68**, and the Boolean functional unit **70** are responsive to the contents of the instruction class field, the opcode field, and the function select field of a present instruction being executed.

1. Integer Operations

For example, when the instruction class, the opcode, and function select indicate that an integer register-to-register addition is to be performed, the integer functional unit may be responsive thereto to perform the indicated operation, while the floating point functional unit and the Boolean functional unit may be responsive thereto to not perform the operation. As will be understood from the cross-referenced applications, however, the floating point functional unit **68** is equipped to perform both floating point and integer operations. Also, the functional units are constructed to each perform more than one instruction simultaneously.

The integer functional unit **66** performs integer functions only. Integer operations typically involve a first source, a second source, and a destination. A given integer instruction will specify a particular operation to be performed on one or more source operands and will specify that the result of the integer operation is to be stored at a given destination. In some instructions, such as address calculations employed in load/store operations, the sources are utilized as a base and an index. The integer functional unit **66** is coupled to a first bus **72** over which the integer functional unit **66** is connected to a switching and multiplexing control (SMC) unit A **74** and an SMC unit B **76**. Each integer instruction executed by the integer functional unit **66** will specify whether each of its sources and destination reside in register set A or register set B.

Suppose that the IEU **10** has received, from the instruction fetch unit (not shown), an instruction to perform an integer register-to-register addition. In various embodiments, the instruction may specify a register bank, perhaps even a separate bank for each source and destination. In one embodiment, the instruction I[] is limited to a thirty-two-bit length, and does not contain any indication of which register bank **16-0** through **16**-n is involved in the instruction. Rather, the bank selector unit **78** controls which register bank is presently active. In one embodiment, the bank selector unit **78** is responsive to one or more bank selection bits in a status word (not shown) within the IEU **10**.

In order to perform the integer addition instruction, the integer functional unit **66** is responsive to the identification in I[14:10] and I[4:0] of the first and second source registers. The integer functional unit **66** places an identification of the first and second source registers at ports S1 and S2, respectively, onto the integer functional unit bus **72** which is coupled to both SMC units A and B **74** and **76**. In one embodiment, the SMC units A and B are each coupled to receive B0–2 from the instruction I[]. In one embodiment, a zero in any respective Bn indicates register set A, and a one indicates register set B. During load/store operations, the source ports of the integer and floating point functional units **66** and **68** are utilized as a base port and an index port, B and I, respectively.

After obtaining the first and second operands from the indicated register sets on the bus **72**, as explained below, the integer functional unit **66** performs the indicated operation upon those operands, and provides the result at port D onto the integer functional unit bus **72**. The SMC units A and B are responsive to B0 to route the result to the appropriate register set A or B.

The SMC unit B is further responsive to the instruction class opcode, and function selection to control whether operands are read from (or results are stored to) either a floating point register RF[] or an integer register RB[]. As indicated, in one embodiment, the registers RF[] may be sixty-four bits wide while the registers are RB[] are only thirty-two bits wide. Thus, SMC unit B controls whether a word or a double word is written to the register set RFB[]. Because all registers within register set A are thirty-two bits wide, SMC unit A need not include means for controlling the width of data transfer on the bus **42**.

All data on the bus **42** are thirty-two bits wide, but other sorts of complexities exist within register set A. The IEU mode integer switch **34** is responsive to the mode control unit **44** of the execution engine **14** to control whether data on the bus **42** are connected through to bus **36**, bus **38** or bus **40**, and vice versa.

IEU mode integer switch **34** is further responsive to I[20:16], I[14:10], and I[4:0]. If a given indicated destination or source is in RA[23:0], the IEU mode integer switch **34** automatically couples the data between lines **42** and **36**. However, for registers RA[31:24], the IEU mode integer switch **34** determines whether data on line **42** is connected to line **38** or line **40**, and vice versa. When interrupts are enabled, IEU mode integer switch **34** connects the SMC unit A to the second subset **28** of integer registers RA[31:24]. When interrupts are disabled, the IEU mode integer switch **34** connects the SMC unit A to the shadow registers RT[31:24]. Thus, an instruction executing within the integer functional unit **66** need not be concerned with whether to address RA[31:24] or RT[31:24]. It will be understood that SMC unit A may advantageously operate identically whether it is being accessed by the integer functional unit **66** or by the floating point functional unit **68**.

2. Floating Point Operations

The floating point functional unit **68** is responsive to the class, opcode, and function select fields of the instruction, to perform floating point operations. The S1, S2, and D ports operate as described for the integer functional unit **66**. SMC unit B is responsive to retrieve floating point operands from, and to write numerical floating point results to, the floating point registers RF[] on bus **52**.

3. Boolean Operations

SMC unit C **80** is responsive to the instruction class, opcode, and function select fields of the instruction I[]. When SMC unit C detects that a comparison operation has been performed by one of the numerical functional units **66** or **68**, it writes the Boolean result over bus **56** to the Boolean register indicated at the D port of the functional unit which performed the comparison.

The Boolean functional unit **70** does not perform comparison instructions as do the integer and floating point functional units **66** and **68**. Rather, the Boolean functional unit **70** is only used in performing bitwise logical combination of Boolean register contents, according to the Boolean functions listed in Table 2.

TABLE 2

Boolean Functions

I[23,22,9,8]	Boolean result calculation
0000	ZERO
0001	S1 AND S2
0010	S1 AND (NOT S2)
0011	S1
0100	(NOT S1) AND S2
0101	S2
0110	S1 XOR 52
0111	S1 OR S2
1000	S1 NOR S2
1001	S1 XNOR S2
1010	NOT S2
1011	S1 OR (NOT S2)
1100	NOT S1
1101	(NOT S1) OR S2
1110	S1 NAND S2
1111	ONE

The advantage which the present invention obtains by having a plurality of homogenous Boolean registers, each of which is individually addressable as the destination of a Boolean operation, will be explained with reference to Tables 3–5. Table 3 illustrates an example of a segment of code which performs a conditional branch based upon a complex Boolean function. The complex Boolean function includes three portions which are OR-ed together. The first portion includes two sub-portions, which are AND-ed together.

TABLE 3

Example of Complex Boolean Function

```
1   RA[1] := 0;
2   IF (((RA[2] = RA[3]) AND (RA[4] > RA[5])) OR
3       (RA[6] < RA[7]) OR
4       (RA[8] <> RA[9])) THEN
5         X()
6   ELSE
7         Y();
8   RA[10] := 1;
```

Table 4 illustrates, in pseudo-assembly form, one likely method by which previous microprocessors would perform the function of Table 3. The code in Table 4 is written as though it were constructed by a compiler of at least normal intelligence operating upon the code of Table 3. That is, the compiler will recognize that the condition expressed in lines 2–4 of Table 3 is passed if any of the three portions is true.

TABLE 4

Execution of Complex Boolean Function Without Boolean Register Set

1	START	LDI	RA[1],0
2	TEST1	CMP	RA[2],RA[3]
3		BNE	TEST2
4		CMP	RA[4],RA[5]
5		BGT	DO_IF
6	TEST2	CMP	RA[6],RA[7]
7		BLT	DO_IF
8	TEST3	CMP	RA[8],RA[9]
9		BEQ	DO_ELSE
10	DO_IF	JSR	ADDRESS OF X()
11		JMP	PAST_ELSE
12	DO_ELSE	JSR	ADDRESS OF Y()
13	PAST_ELSE	LDI	RA[10],1

The assignment at line 1 of Table 3 is performed by the "load immediate" statement at line 1 of Table 4. The first portion of the complex Boolean condition, expressed at line 2 of Table 3, is represented by the statements in lines 2–5 of Table 4. To test whether RA[2] equals RA[3], the compare statement at line 2 of Table 4 performs a subtraction of RA[2] from RA [3] or vice versa, depending upon the implementation, and may or may not store the result of that subtraction. The important function performed by the comparison statement is that the zero, minus, and carry flags will be appropriately set or cleared.

The conditional branch statement at line 3 of Table 4 branches to a subsequent portion of code upon the condition that RA[2] did not equal RA[3]. If the two were unequal, the zero flag will be clear, and there is no need to perform the second sub-portion. The existence of the conditional branch statement at line 3 of Table 4 prevents the further fetching, decoding, and executing of any subsequent statement in Table 4 until the results of the comparison in line 2 are known, causing a pipeline stall. If the first sub-portion of the first portion (TEST1) is passed, the second sub-portion at line 4 of Table 4 then compares RA[4]to RA[5], again setting and clearing the appropriate status flags.

If RA[2] equals RA[3], and RA[4] is greater than RA[5], there is no need to test the remaining two portions (TEST2 and TEST3) in the complex Boolean function, and the statement at Table 4, line 5, will conditionally branch to the label DO_IF, to perform the operation inside the "IF" of Table 3. However, if the first portion of the test is failed, additional processing is required to determine which of the "IF" and "ELSE" portions should be executed.

The second portion of the Boolean function is the comparison of RA[6] to RA[7], at line 6 of Table 4, which again sets and clears the appropriate status flags. If the condition "less than" is indicated by the status flags, the complex Boolean function is passed, and execution may immediately branch to the DO_IF label. In various prior microprocessors, the "less than" condition may be tested by examining the minus flag. If RA[7] was not less than RA[6], the third portion of the test must be performed. The statement at line 8 of Table 4 compares RA[8] to RA[9]. If this comparison is failed, the "ELSE" code should be executed; otherwise, execution may simply fall through to the "IF" code at line 10 of Table 4, which is followed by an additional jump around the "ELSE" code. Each of the conditional branches in Table 4, at lines 3, 5, 7 and 9, results in a separate pipeline stall, significantly increasing the processing time required for handling this complex Boolean function.

The greatly improved throughput which results from employing the Boolean register set C of the present invention will now readily be seen with specific reference to Table 5.

TABLE 5

Execution of Complex Boolean Function With Boolean Register Set

1	START	LDI	RA[1],0
2	TEST1	CMP	RC[11],RA[2],RA[3],EQ
3		CMP	RC[12],RA[4],RA[5],GT
4	TEST2	CMP	RC[13],RA[6],RA[7],LT
5	TEST3	CMP	RC[14],RA[8],RA[9],NE
6	COMPLEX	AND	RC[15],RC[11],RC[12]
7		OR	RC[16],RC[13],RC[14]
8		OR	RC[17],RC[15],RC[16]
9		BC	RC[17],DO_ELSE
10	DO_IF	JSR	ADDRESS OF X()
11		JMP	PAST_ELSE
12	DO_ELSE	JSR	ADDRESS OF Y()
13	PAST_ELSE	LDI	RA[10],1

Most notably seen at lines 2–5 of Table 5, the Boolean register set C allows the microprocessor to perform the three test portions back-to-back without intervening branching. Each Boolean comparison specifies two operands, a destination, and a Boolean condition for which to test. For example, the comparison at line 2 of Table 5 compares the contents of RA[2] to the contents of RA[3], tests them for equality, and stores into RC[11] the Boolean value of the result of the comparison. Note that each comparison of the Boolean function stores its respective intermediate results in a separate Boolean register. As will be understood with reference to the above-referenced related applications, the IEU **10** is capable of simultaneously performing more than one of the comparisons.

After at least the first two comparisons at lines 2–3 of Table 5 have been completed, the two respective comparison results are AND-ed together as shown at line 6 of Table 3. RC[15] then holds the result of the first portion of the test. The results of the second and third sub-portions of the Boolean function are OR-ed together as seen in Table 5, line 7. It will be understood that, because there are no data dependencies involved, the AND at line 6 and the OR-ed in line 7 may be performed in parallel. Finally, the results of those two operations are OR-ed together as seen at line 8 of Table 5. It will be understood that register RC[17] will then contain a Boolean value indicating the truth or falsity of the entire complex Boolean function of Table 3. It is then possible to perform a single conditional branch, shown at line 9 of Table 5. In the mode shown in Table 5, the method branches to the "ELSE" code if Boolean register RC[17] is clear, indicating that the complex function was failed. The remainder of the code may be the same as it was without the Boolean register set as seen in Table 4.

The Boolean functional unit **70** is responsive to the instruction class, opcode, and function select fields as are the other functional units. Thus, it will be understood with reference to Table 5 again, that the integer and/or floating point functional units will perform the instructions in lines 1–5 and 13, and the Boolean functional unit **70** will perform the Boolean bitwise combination instructions in lines 6–8. The control flow and branching instructions in line 9–12 will be performed by elements of the IEU **10** which are not shown in FIG. **1**.

III. DATA PATHS

FIGS. **2–5** illustrate further details of the data paths within the floating point, integer, and Boolean portions of the IEU, respectively.

A. Floating Point Portion Data Paths

As seen in FIG. **2**, the register set FB **20** is a multi-ported register set. In one embodiment, the register set FB **20** has two write ports WFB0–1, and five read ports RDFB0–4. The-floating point functional unit **68** of FIG. **1** is comprised of the ALU2 **102**, FALU **104**, MULT **106**, and NULL **108** of FIG. **2**. All elements of FIG. **2** except the register set **20** and the elements **102–108** comprise the SMC unit B of FIG. **1**.

External, bidirectional data bus EX_DATA[] provides data to the floating point load/store unit **122**. Immediate floating point data bus LDF_IMED[] provides data from a "load immediate" instruction. Other immediate floating point data are provided on busses $RFF1_{13}$ IMED and RFF2_IMED, such as is involved in an "add immediate" instruction. Data are also provided on bus EX_SR_DT[], in response to a "special register move" instruction. Data may also arrive from the integer portion, shown in FIG. **3**, on busses **114** and **120**.

The floating point register set's two write ports WFB0 and WFB1 are coupled to write multiplexers **110-0** and **110-1**, respectively. The write multiplexers **110** receive data from: the ALU0 or SHF0 of the integer portion of FIG. **3**; the FALU; the MULT; the ALU2; either EX_SR_DT[] or LDF_IMED[]; and EX_DATA[]. Those skilled in the art will understand that control signals (not shown) determine which input is selected at each port, and address signals (not shown) determine to which register the input data are written. Multiplexer control and register addressing are within the skill of persons in the art, and will not be discussed for any multiplexer or register set in the present invention.

The floating point register set's five read ports RDFB0 to RDFB4 are coupled to read multiplexers **112-0** to **112-4**, respectively. The read multiplexers each also receives data from: either EX_SR_DT[] or LDF_IMED[], on load immediate bypass bus **126**; a load external data bypass bus **127**, which allows external load data to skip the register set FB; the output of the ALU2 **102**, which performs non-multiplication integer operations; the FALU **104**, which performs non-multiplication floating point operations; the

MULT **106**, which performs multiplication operations; and either the ALU0 **140** or the SHF0 **144** of the integer portion shown in FIG. **3**, which respectively perform non-multiplication integer operations and shift operations. Read multiplexers **112-1** and **112-3** also receive data from RFF1_IMED[] and RFF2_IMED[], respectively.

Each arithmetic-type unit **102–106** in the floating point portion receives two inputs, from respective sets of first and second source multiplexers S1 and S2. The first source of each unit ALU2, FALU, and MULT comes from the output of either read multiplexer **112-0** or **112-2**, and the second source comes from the output of either read multiplexer **112-1** or **112-3**. The sources of the FALU and the MULT may also come from the integer portion of FIG. **3** on bus **114**.

The results of the ALU2, FALU, and MULT are provided back to the write multiplexers **110** for storage into the floating point registers RF[], and also to the read multiplexers **112** for re-use as operands of subsequent operations. The FALU also outputs a signal FALU_BD indicating the Boolean result of a floating point comparison operation. FALU_BD is calculated directly from internal zero and sign flags of the FALU.

Null byte tester NULL **108** performs null byte testing operations upon an operand from a first source multiplexer, in one mode that of the ALU2. NULL **108** outputs a Boolean signal NULLB_BD indicating whether the thirty-two-bit first source operand includes a byte of value zero.

The outputs of read multiplexers **112-0**, **112-1**, and **112-4** are provided to the integer portion (of FIG. **3**) on bus **118**. The output of read multiplexer **112-4** is also provided as STDT_FP[] store data to the floating point load/store unit **122**.

FIG. **5** illustrates further details of the control of the S1 and S2 multiplexers. As seen, in one embodiment, each S1 multiplexer may be responsive to bit B1 of the instruction I[], and each S2 multiplexer may be responsive to bit B2 of the instruction I[]. The S1 and S2 multiplexers select the sources for the various functional units. The sources may come from either of the register files, as controlled by the B1 and B2 bits of the instruction itself. Additionally, each register file includes two read ports from which the sources may come, as controlled by hardware not shown in the FIGS.

B. Integer Portion Data Paths

As seen in FIG. **3**, the register set A **18** is also multi-ported. In one embodiment, the register set A **18** has two write ports WA0–1, and five read ports RDA0–4. The integer functional unit **66** of FIG. **1** is comprised of the ALU0 **140**, ALU1 **142**, SHF0 **144**, and NULL **146** of FIG. **3**. All elements of FIG. **3** except the register set **18** and the elements **140–146** comprise the SMC unit A of FIG. **1**.

External data bus EX_DATA[] provides data to the integer load/store unit **152**. Immediate integer data on bus LDI_IMED[] are provided in response to a "load immediate" instruction. Other immediate integer data are provided on busses RFA1_IMED and RFA2_IMED in response to non-load immediate instructions, such as an "add immediate". Data are also provided on bus EX_SR_DT[] in response to a "special register move" instruction. Data may also arrive from the floating point portion (shown in FIG. **2**) on busses **116** and **118**.

The integer register set's two write ports WA0 and WA1 are coupled to write multiplexers **148-0** and **148-1**, respectively. The write multiplexers **148** receive data from: the FALU or MULT of the floating point portion (of FIG. **2**); the ALU0; the ALU1; the SHF0; either EX_SR_DT[] or LDI_IMED[]; and EX_DATA[].

The integer register set's five read ports RDA0 to RDA4 are coupled to read multiplexers **150-0** to **150-4**, respectively. Each read multiplexer also receives data from: either EX_SR_DT[] or LDI_IMED[] on load immediate bypass bus **160**; a load external data bypass bus **154**, which allows external load data to skip the register set A; ALU0; ALU1; SHF0; and either the FALU or the MULT of the floating point portion (of FIG. **2**). Read multiplexers **150-1** and **150-3** also receive data from RFA1_IMED[] and RFA2_IMED[], respectively.

Each arithmetic-type unit **140–144** in the integer portion receives two inputs, from respective sets of first and second source multiplexers S1 and S2. The first source of ALU0 comes from either the output of read multiplexer **150-2**, or a thirty-two-bit wide constant zero (0000_{hex}) or floating point read multiplexer **112-4**. The second source of ALU0 comes from either read multiplexer **150-3** or floating point read multiplexer **112-1**. The first source of ALU1 comes from either read multiplexer **150-0** or IF_PC[]. IF_PC[] is used in calculating a return address needed by the instruction fetch unit (not shown), due to the IEU's ability to perform instructions in an out-of-order sequence. The second source of ALUL comes from either read multiplexer **150-1** or CF_OFFSET[]. CF_OFFSET[] is used in calculating a return address for a CALL instruction, also due to the out-of-order capability.

The first source of the shifter SHF0 **144** is from either: floating point read multiplexer **112-0** or **112-4**; or any integer read multiplexer **150**. The second source of SHF0 is from either: floating point read multiplexer **112-0** or **112-4**; or integer read multiplexer **150-0**, **150-2**, or **150-4**. SHF0 takes a third input from a shift amount multiplexer (SA). The third input controls how far to shift, and is taken by the SA multiplexer from either: floating point read multiplexer **112-1**; integer read multiplexer **150-1** or **150-3**; or a five-bit wide constant thirty-one (11111_2 or 31_{10}). The shifter SHF0 requires a fourth input from the size multiplexer (S). The fourth input controls how much data to shift, and is taken by the S multiplexer from either: read multiplexer **150-1**; read multiplexer **150-3**; or a five-bit wide constant sixteen (10000_2 or 16_{10}).

The results of the ALU0, ALU1, and SHF0 are provided back to the write multiplexers **148** for storage into the integer registers RA[], and also to the read multiplexers **150** for re-use as operands of subsequent operations. The output of either ALU0 or SHF0 is provided on bus **120** to the floating point portion of FIG. **3**. The ALU0 and ALU1 also output signals ALU0_BD and ALU1_BD, respectively, indicating the Boolean results of integer comparison operations. ALU0_BD and ALU1_BD are calculated directly from the zero and sign flags of the respective functional units. ALU0 also outputs signals EX_TADR[] and EX_VM_ADR. EX_TADR[] is the target address generated for an absolute branch instruction, and is sent to the IFU (not shown) for fetching the target instruction. EX_VM_ADR[] is the virtual address used for all loads from memory and stores to memory, and is sent to the VMU (not shown) for address translation.

Null byte tester NULL **146** performs null byte testing operations upon an operand from a first source multiplexer. In one embodiment, the operand is from the ALU0. NULL **146** outputs a Boolean signal NULLA_BD indicating whether the thirty-two-bit first source operand includes a byte of value zero.

The outputs of read multiplexers **150-0** and **150-1** are provided to the floating point portion (of FIG. **2**) on bus **114**. The output of read multiplexer **150-4** is also provided as STDT_INT[] store data to the integer load/store unit **152**.

A control bit PSR[7] is provided to the register set A **18**. It is this signal which, in FIG. **1**, is provided from the mode control unit **44** to the IEU mode integer switch **34** on line **46**. The IEU mode integer switch is internal to the register set A **18** as shown in FIG. **3**.

FIG. **6** illustrates further details of the control of the S1 and S2 multiplexers. The signal ALU0_BD

C. Boolean Portion Data Paths

As seen in FIG. **4**, the register set C **22** is also multi-ported. In one embodiment, the register set C **22** has two write-ports WC0–1, and five read ports RDA0–4. All elements of FIG. **4** except the register set **22** and the Boolean combinational unit **70** comprise the SMC unit C of FIG. **1**.

The Boolean register set's two write ports WC0 and WC1 are coupled to write multiplexers **170-0** and **170-1**, respectively. The write multiplexers **170** receive data from the output of the Boolean combinational unit **70**, indicating the Boolean result of a Boolean combinational operation; ALU0_BD from the integer portion of FIG. 3, indicating the Boolean result of an integer comparison; FALU_BD from the floating point portion of FIG. 2, indicating the Boolean result of a floating point comparison; either ALU1_BD_P from ALU1, indicating the results of a compare instruction in ALU1, or NULLA_BD from NULL **146**, indicating a null byte in the integer portion; and either ALU2_BD_P from ALU2, indicating the results of a compare operation in ALU2, or NULLB_BD from NULL **108**, indicating a null byte in the floating point portion. In one mode, the ALU0_BD, ALU1_BD, ALU2_BD, and FALU_BD signals are not taken from the data paths, but are calculated as a function of the zero flag, minus flag, carry flag, and other condition flags in the PSR. In one mode, wherein up to eight instructions may be executing at one instant in the IEU, the IEU maintains up to eight PSRs.

The Boolean register set C is also coupled to bus EX_SR_DT[], for use with "special register move" instructions. The CSR may be written or read as a whole, as though it were a single thirty-two-bit register. This enables rapid saving and restoration of machine state information, such as may be necessary upon certain drastic system errors or upon certain forms of grand scale context switching.

The Boolean register set's five read ports RDC0 to RDC3 are coupled to read multiplexers **172-0** to **172-4**, respectively. The read multiplexers **172** receive the same set of inputs as the write multiplexers **170** receive. The Boolean combinational unit **70** receives inputs from read multiplexers **170-0** and **170-1**. Read multiplexers **172-2** and **172-3** respectively provide signals BLBP_CPORT and BLBP_DPORT. BLBP_CPORT is used as the basis for conditional branching instructions in the IEU. BLBP_DPORT is used in the "add with Boolean" instruction, which sets an integer register in the A or B set to zero or one (with leading zeroes), depending upon the content of a register in the C set. Read port RDC4 is presently unused, and is reserved for future enhancements of the Boolean functionality of the IEU.

IV. CONCLUSION

While the features and advantages of the present invention have been described with respect to particular embodiments thereof, and in varying degrees of detail, it will be appreciated that the invention is not limited to the described embodiments. The following Claims define the invention to be afforded patent coverage.

We claim:

1. In a data processing system, which includes a central processing unit (CPU) that performs operations by executing instructions, a data register system comprising:

a first register set including a plurality of first registers each for holding integer data;

a second register set including a plurality of second registers each for holding integer data or floating point data, wherein a specific instruction includes a field specifying which of said first and second register sets is to be accessed in response to execution of said specific instruction; and

means, responsive to the field, for accessing said first register set or said second register set as specified by said field, including

i) reading means for reading an operand value from either the first register set or second register set as specified by said field, and

ii) writing mean for writing a result value to sad first register set or said second register set as specified by said field.

2. The apparatus of claim **1**, wherein said first and second register sets each have two write ports and five read ports.

3. The apparatus of claim **1**, further comprising execution means for executing said specific instruction, wherein said specific instruction performs an operation upon operands to generate a result, said specific instruction specifying a respective source address for each operand and a destination address for the result of said specific instruction.

4. The apparatus of claim **1**, wherein said specific instruction can specify a first and a second source address and a destination address, with each address specifying either of the first or second register sets such that said specific instruction requires access to both register sets.

5. The apparatus of claim **4**, wherein said means for accessing provides said specific instruction parallel access to both the first and second register sets.

6. An apparatus, for use with a data processing system that performs read operations and write operations upon data values of a first data type and a first data width, wherein the first data type is floating point, and upon data values of a second data type and a second data width different from the first data width, the second data type is integer, the data processing system specifying a read address and data type for each read and a write address and data content for each write, the apparatus comprising:

a register set including a plurality of individually addressable registers, each register being wide enough to hold a value of the first data type or the second data type;

read access means, responsive to the data processing system performing a given read operation of a specific data type, for accessing said register set to retrieve data from a given register, which is individually addressed at a specified read address of said given read operation,

write access means, responsive to the data processing system performing a given write operation, for accessing said register set to store into a given register, which is individually addressed at the specified write address of said given write operation, data specified by said write operation; and

wherein said read and write access means, respectively, retrieve and store data having the first data width responsive to the data processing system performing floating point operations, and data having the second data width responsive to the data processing system performing integer operations.

7. The apparatus of claim **6**, wherein the first data width is sixty-four bits and the second data width is thirty-two bits.

8. The apparatus of claim **6**, further comprising processing means for executing instructions including Boolean

execution unit to execute Boolean combinational instructions each operating on one or more Boolean operands to generate a Boolean result, each Boolean combinational instruction including one or more Boolean fields specifying a location of each operand and result, an integer execution unit to execute integer instructions each operating on one or more integer operands to generate an integer result, each integer instruction including one or more integer fields specifying a location of each operand and result, and a floating point execution unit to execute floating point instructions each operating on one or more floating point operands to generate a floating point result, each floating point instruction including one or more floating point fields specifying a location of each operand and result.

9. The apparatus of claim **8**, further comprising a Boolean register set having a plurality of Boolean registers, each Boolean register for holding one of said Boolean operands or Boolean result.

10. The apparatus of claim **9**, wherein said plurality of Boolean registers include:

(i) a first set of Boolean registers, and

(ii) a second set of Boolean registers;

means, coupled to said plurality of Boolean registers, for selecting said first or said second set of Boolean registers as a currently active set,

wherein said Boolean execution unit is responsive to said means for selecting and stores results into only said currently active set of said Boolean registers; and

means, responsive to execution of a given Boolean instruction by said Boolean execution unit, for storing the result of said given Boolean instruction into one of said Boolean registers, said one Boolean register being indicated by said given Boolean instruction as the destination of its Boolean result.

11. The apparatus of claim **8**, wherein said Boolean execution unit comprises: numerical execution means for executing numerical comparison instructions to compare two multi-bit numerical operands and to accordingly produce a single-bit Boolean value result.

12. An apparatus comprising:

integer execution means for executing integer instructions, each integer instruction performing an integer operation upon one or more integer value operands and generating an integer value result;

floating point execution means for executing floating point instructions, each floating point operation performing a floating point operation upon one or more floating point value operands and generating a floating point value result;

boolean execution means for executing boolean instructions, each boolean operation performing a boolean operation upon one or more boolean value operands and generating a boolean value result;

wherein each instruction specifies one or more sources from which its one or more operands are to be retrieved and further specifies a destination to which its result is to be stored, each operation also optionally specifying an integer value base and an integer value index;

a register bank including,

i) first register set, having a plurality of first registers, for holding integer values and floating point values;

ii) second register set, having a plurality of second registers, for holding integer values; and

iii) third register set, having a plurality of third registers, for holding Boolean values;

access means, coupled to said first register set, said second register set, said third register set and to all three execution means, for,

i) retrieving, from any one first register, an integer value operand for the integer execution means, a floating point value operand for the floating point execution means, or an integer value base or index for either execution means, as indicated by an instruction;

ii) storing, into any one first register, an integer value result from the integer execution means or a floating point value result from the floating point execution means, as indicated by an instruction;

iii) retrieving, from any one second register, an integer value operand for said integer execution means, or an integer value base or index for either execution means, as indicated by an instruction;

iv) storing, into any one second register, an integer value result from said integer execution means, as indicated by an instruction;

v) retrieving, from any one third register, a Boolean value operand for the Boolean execution means, as indicated by a Boolean combinational instruction, and

vi) storing, into any one third register, a Boolean value result from the Boolean execution means, as indicated by a Boolean combinational instruction.

13. An instruction execution unit of a RISC processor, comprising:

an execution engine that includes a data dependency checker, an integer functional unit, a floating point functional unit, a boolean functional unit, means for performing a context switch, and a mode control unit that provides mode information;

a register file, connected to said execution engine, having two or more register banks, each bank having,

an integer register set that includes, a first integer register set that includes a first subset of registers and a second subset of registers, a shadow integer register set and mode switch, wherein a first register in said first integer register set is set to zero, wherein said mode switch is switchingly coupled to said mode control unit, said first integer register set and said shadow integer register set, wherein said mode switch provides access to said second subset of registers or said shadow integer register set depending upon said mode information, whereby all access to said integer register set is via said mode switch,

a re-typable register set, wherein said re-typable register set can store floating point data or integer data, said re-typable register set including means for preventing accidental access of a floating point value as though it is an integer value or accidental access of a integer value as though it is a floating point value, wherein a first register in said re-typable register set is set to zero, and

a boolean register set that includes a condition status register (CSR) that includes a plurality of boolean registers, a previous condition status register (PCSR) that includes a plurality of boolean registers and a selector unit that is responsive to said mode information to select between said CSR and said PCSR, wherein a first boolean register in said CSR is set to zero,

wherein said data dependency checker allows a slave instruction to execute without delay when the result of

a master instruction is said first register in said first integer register set.

14. The instruction execution unit of claim 13, wherein said two or more register banks comprise the same hardware configuration.

15. The instruction execution unit of claim 13, wherein said mode control unit is a processor status register that includes a flag to indicate whether interrupts are enable or disabled.

16. The instruction execution unit of claim 13, wherein said CSR is used when interrupts are enabled and said PCSR is used when interrupts are disabled.

17. The instruction execution unit of claim 13, wherein said PCSR is available to said execution engine as a special register.

18. The instruction execution unit of claim 13, wherein said selector unit writes the contents of said CSR into said PCSR when interrupts are disabled, overwriting the former contents of said PCSR, and when interrupts are re-enabled, said selector unit writes the contents of said PCSR into said CSR.

19. The instruction execution unit of claim 13, wherein said boolean register set does not include dedicated condition flags.

20. The instruction execution unit of claim 13, wherein said execution engine operates on one or more instructions having a destination field that specifies said integer register set or said re-typable register set, an opcode, a first operand field that specifies said integer register set or said re-typable register set, and a second operand field that specifies said integer register set or said re-typable register set.

21. The instruction execution unit of claim 13, wherein said integer functional unit and said floating point functional unit are configured to execute a comparison instruction, wherein a destination of said comparison instruction is said boolean register set.

22. The instruction execution unit of claim 13, wherein said instruction further includes a field indicating an addressing mode.

23. The instruction execution unit of claim 13, wherein said floating point functional unit is configured to perform floating point and integer operations.

24. The instruction execution unit of claim 13, wherein said integer functional unit, said floating point functional unit and said boolean functional unit are each configured to execute a plurality of instructions simultaneously.

25. The instruction execution unit of claim 13, further comprising a switching and multiplexing control unit connected between said integer functional unit and said register file, wherein said integer functional unit executes instructions having at least one source and at least one destination, each instruction indicating whether said at least one source and said at least one destination reside in said integer register set or said re-typable register set.

26. The instruction execution unit of claim 25, wherein said mode switch connects said switching and multiplexing control unit to said shadow integer register set when interrupts are disabled.

27. The instruction execution unit of claim 26, wherein said mode control unit is a processor status register that includes a flag that indicates whether interrupts are enabled or disabled.

28. The instruction execution unit of claim 13, further comprising a switching and multiplexing control unit connected between said floating point functional unit and said register file, wherein said floating point functional unit executes instructions having at least one source and at least one destination, each instruction indicating whether said at least one source and said at least one destination reside in said integer register set or said re-typable register set.

29. The instruction execution unit of claim 13, wherein said execution engine executes an instruction having at least one source and at least one destination, wherein said instruction further includes a field that indicates which of said two or more banks has stored therein said one or more source and where said destination is located within said two or more banks.

30. The instruction execution unit of claim 13, further includes a bank selector unit that selects between said two or more register banks.

* * * * *

EXHIBIT G

US006044449A

United States Patent [19]

Garg et al.

[11] **Patent Number: 6,044,449**

[45] **Date of Patent: *Mar. 28, 2000**

[54] **RISC MICROPROCESSOR ARCHITECTURE IMPLEMENTING MULTIPLE TYPED REGISTER SETS**

[75] Inventors: **Sanjiy Garg**, Fremont; **Derek J. Lentz**, Los Gatos; **Le Trong Nguyen**, Monte Sereno; **Sho Long Chen**, Saratoga, all of Calif.

[73] Assignee: **Seiko Epson Corporation**, Tokyo, Japan

[*] Notice: This patent is subject to a terminal disclaimer.

[21] Appl. No.: **09/188,708**

[22] Filed: **Nov. 10, 1998**

Related U.S. Application Data

[63] Continuation of application No. 08/937,361, Sep. 25, 1997, Pat. No. 5,838,986, which is a continuation of application No. 08/665,845, Jun. 19, 1996, Pat. No. 5,682,546, which is a continuation of application No. 08/465,239, Jun. 5, 1995, Pat. No. 5,560,035, which is a continuation of application No. 07/726,773, Jul. 8, 1991, Pat. No. 5,493,687.

[51] **Int. Cl.**^{7} **G06F 15/00**

[52] **U.S. Cl.** **712/23**; 712/278

[58] **Field of Search** 712/23, 24, 208, 712/212, 217, 218, 219, 228

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,212,076	7/1980	Conners	364/706
4,626,989	12/1986	Torii	364/200
4,675,806	6/1987	Uchida	364/200
4,722,049	1/1988	Lahti	364/200
4,807,115	2/1989	Torng	364/200

(List continued on next page.)

FOREIGN PATENT DOCUMENTS

0 170 284	2/1986	European Pat. Off. .
0 213 843	3/1987	European Pat. Off. .
0 241 909	10/1987	European Pat. Off. .
0 454 636	10/1991	European Pat. Off. .
2 190 521	11/1987	United Kingdom .

OTHER PUBLICATIONS

Adams et al. Utilising Low Level Parallelism in General Purpose Code: Harp Project pp. 12–24, Oct. 1990.

Smith et al., "Implementation of Precise Interrupts in Pipelined Processors," *Proceedings of the 12th Annual International Sympsium on Computer Architecture*, Jun. 1985, pp. 36–44.

Wedig, R.G., *Detection of Concurrency In Directly Executed Language Instrtuction Streams*, (Dissertation), Jun. 1982, pp. 1–179.

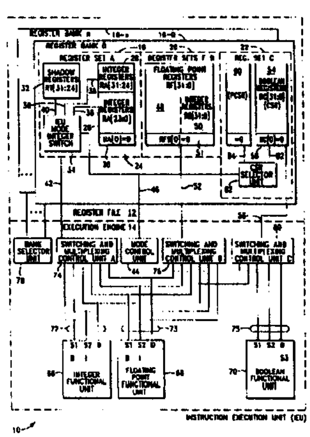
(List continued on next page.)

Primary Examiner—Larry D. Donaghue
Attorney, Agent, or Firm—Sterne, Kessler, Goldstein & Fox P.L.L.C.

[57] **ABSTRACT**

A register system for a data processor which operates in a plurality of modes. The register system provides multiple, identical banks of register sets, the data processor controlling access such that instructions and processes need not specify any given bank. An integer register set includes first (RA[23:0]) and second (RA[31:24]) subsets, and a shadow subset (RT[31:24]). While the data processor is in a first mode, instructions access the first and second subsets. While the data processor is in a second mode, instructions may access the first subset, but any attempts to access the second subset are re-routed to the shadow subset instead, transparently to the instructions, allowing system routines to seemingly use the second subset without having to save and restore data which user routines have written to the second subset. A re-typable register set provides integer width data and floating point width data in response to integer instructions and floating point instructions, respectively. Boolean comparison instructions specify particular integer or floating point registers for source data to be compared, and specify a particular Boolean register for the result, so there are no dedicated, fixed-location status flags. Boolean combinational instructions combine specified Boolean registers, for performing complex Boolean comparisons without intervening conditional branch instructions, to minimize pipeline disruption.

12 Claims, 9 Drawing Sheets



U.S. PATENT DOCUMENTS

5,125,092	6/1992	Prener	395/725
5,201,056	4/1993	Daniel et al.	395/800
5,226,126	7/1993	McFarland et al.	395/375
5,230,068	7/1993	Van Dyke et al.	395/375
5,241,636	8/1993	Kohn	712/23
5,442,757	8/1995	McFarland et al.	395/375
5,487,156	1/1996	Popescu et al.	395/375
5,493,687	2/1996	Garg et al.	712/23
5,560,035	9/1996	Garg et al.	395/800.23
5,561,776	10/1996	Popescu et al.	395/375
5,592,636	1/1997	Popescu et al.	395/586
5,625,837	4/1997	Popescu et al.	395/800
5,627,983	5/1997	Popescu et al.	395/393
5,682,546	10/1997	Garg et al.	395/800.23
5,708,841	1/1998	Popescu et al.	355/800
5,768,575	6/1998	McFarland et al.	395/569
5,797,025	8/1998	Popescu et al.	395/800
5,832,293	11/1998	Popescu et al.	395/800.23
5,838,586	11/1998	Garg et al.	712/23
5,838,986	11/1998	Garg et al.	395/800.23

OTHER PUBLICATIONS

Agerwala et al., "High Performance Reduced Instruction Set Processors," IBM Research Division, Mar. 31, 1987, pp. 1–61.

Gross et al., "Optimizing Delayed Branches," *Proceedings of the 5th Annual Workshop on Microprogramming*, Oct. 5–7, 1982, pp. 114–120.

Tjaden et al., "Representation of Concurrency with Ordering Matrices," *IEEE Trans. On Computers*, vol. C–22, No. 8, Aug. 1973, pp. 752–761.

Tjaden, *Representation and Detection of Concurrency Using Ordering Matrices*, (Dissertation), 1972, pp. 1–199.

Foster et al., "Percolation of Code to Enhance Parallel Dispatching and Execution," *IEEE Trans. On Computers*, Dec. 1971, pp. 1411–1415.

Thornton, J.E., *Design of a Computer: The Control Data 6600*, Control Data Corporation, 1970, pp. 58–140.

Weiss et al., "Instruction Issue Logic in Pipelined Supercomputers," Reprinted from *IEEE Trans. on Computers*, vol. C–33, No. 11, Nov. 1984, pp. 1013–1022.

Tomasulo, R.M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal*, vol. 11, Jan. 1967, pp. 25–33.

Tjaden et al., "Detection and Parallel Execution of Independent Instructions," *IEEE Trans. On Computers*, vol. C–19, No. 10, Oct. 1970, pp. 889–895.

Smith et al., "Limits on Multiple Instruction Issue," *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 1989, pp. 290–302.

Pleszkun et al., "The Performance Potential of Multiple Functional Unit Processors," *Proceedings of the 15th Annual Symposium on Computer Architecture*, Jun. 1988, pp. 37–44.

Pleszkun et al., "WISQ: A Restartable Architecture Using Queues," *Proceedings of the 14th International Symposium on Computer Architecture*, Jun. 1987, pp. 290–299.

Patt et al., "Critical Issues Regarding HPS, A High Performance Microarchitecture," *Proceedings of the 18th Annual Workshop on Microprogramming*, Dec. 1985, pp. 109–116.

Hwu et al., "Checkpoint Repair for High–Performance Out–of–Order Execution Machines," *IEEE Trans. On Computers*, vol. C–36, No. 12, Dec. 1987, pp. 1496–1514.

Patt et al., "HPS, A New Microarchitecture: Rationale and Introduction," *Proceedings of the 18th Annual Workshop on Microprogramming*, Dec. 1985, pp. 103–108.

Keller, R.M., "Look–Ahead Processors," *Computing Surveys*, vol. 7, No. 4, Dec. 1975, pp. 177–195.

Jouppi et al., "Avaialable Instruction–Level Parallelism for Superscalar and Superpipelined Machines," *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 1989, pp. 272–282.

Hwu et al., "HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality," *Proceedings from ISCA–13*, Tokyo, Japan, Jun. 2–5, 1986, pp. 297–306.

Hwu et al., "Exploiting Parallel Microprocessor Microarchitectures with a Compiler Code Generator," *Proceedings of the 15th Annual Symposium on Computer Architecture*, Jun. 1988, pp. 45–53.

Colwell et al., "A VLIW Architecture for a Trace Scheduling Compiler," *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1987, pp. 180–192.

Uht, A.K., "An Efficient Hardware Algorithm to Extract Concurrency From General–Purpose Code," *Proceedings of the 19th Annual Hawaii International Conference on System Sciences*, 1986, pp. 41–50.

Charlesworth, A.E., "An Approach to Scientific Array Processing: The Architectural Design of the AP–120B/FPS–164 Family," *Computer*, vol. 14, Sep. 1981, pp. 18–27.

Acosta, Ramón D. et al., "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors," *IEEE Transactions On Computers*, vol. C–35, No. 9, Sep. 1986, pp. 815–828.

Johnson, William M., *Super–Scalar Processor Design*, (Dissertation), Copyright 1989, 134 pages.

Sohi, Gurindar S. and Sriram Vajapeyam, "Instruction Issue Logic For High–Performance, Interruptable Pipelined Processors," *Conference Proceedings of the 14th Annual International Symposium on Computer Architecture*, Jun. 2–5, 1987, pp. 27–34.

Smith, M.D. et al., "Boosting Beyond Static Scheduling in a Superscalar Processor," *IEEE*, 1990, pp. 344–354.

Murakami, K. et al., "SIMP (Single Instruction stream/ Multiple instruction Pipelining): A Novel High–Speed Single–Processor Architecture," *ACM*, 1989, pp. 78–85.

Jouppi, N.P., "The Nonuniform Distribution of Instruction––Level and Machine Parallelism and Its Effect on Performance," *IEEE Transactions on Computers*, vol. 38, No. 12, Dec. 1989, pp. 1645–1658.

Horst, R.W. et al., "Multiple Instruction Issue in the Non-Stop Cyclone Processor," *IEEE*, 1990, pp. 216–226.

Goodman, J.R. and Hsu, W., "Code Scheduling and Register Allocation in Large Basic Blocks," *ACM*, 1988, pp. 442–452.

Lam, M.S., "Instruction Scheduling For Superscalar Architectures," *Annu. Rev. Comput. Sci.*, vol. 4, 1990, pp. 173–201.

Aiken, A. and Nicolau, A., "Perfect Pipelining: A New Loop Parallelization Technique*," pp. 221–235.

Jouppi, N.H., "Integration and Packaging Plateaus of Processor Performance," *IEEE*, 1989, pp. 229–232.

Patterson et al., "A VLSI RISC," *IEEE Computer*, vol. 15, No. 9, pp. 8–18, Sep. 1982.

Maejima et al., "A 16–bit Microprocessor with Multi–Register Bank Architecture," *Proc. Fall Joint Computer Conference*, Nov. 2–6, 1986, pp. 1014–1019.

Birman et al., "Design of a High–Speed Arithmetic Datapath," *IEEE*, pp. 214–216, 1988.

Ruby B. Lee, "Precision Architecture," *IEEE Computer*, pp. 78–91, Jan. 1989.

Molnar et al., "Floating–Point Processors," *IEEE Intl. Solid–State Circuits Conf.*, pp. 48–49, plus Figure 1, Feb. 1989.

Steven et al., "Harp: A Parallel Pipelined RISC Processor," *Microprocessors and Microsystems*, vol. 13, No. 9, pp. 579–586, Nov. 1989.

Groves et al., "An IBM Second Generation RISC Processor Architecture," *35th IEEE Computer Society International Conference*, Feb. 26, 1990, pp. 166–172.

Miller et al., "Exploiting Large Register Sets," *Microprocessors and Microsystems*, vol. 14, No. 6, Jul. 1990, pp. 333–340.

Adams et al., "Utilizing Low Level Parallelism in General Purpose Code: The HARP Project," *Micrprocessing and Micropragramming*, vol. 29, No. 3, Oct. 1990, pp. 137–149.

Daryl Odnert et al., "Architecture and Computer Enhancements for PA–RISC Workstations," *Proc. from IEEE Compcon*, San Francisco, CA, pp. 214–218, feb. 1991.

Colin Hunter, "*Series 3200 Programmer's Reference Manual*," Prentice–Hall Inc., Englewood Cliffs, NJ, 1987, pp. 2–4, 2–21, 2–23, 6–14, and 6–126.

IBM Journal of Research and Development, vol. 34, No. 1, Jan. 1990, pp. 1–70.-

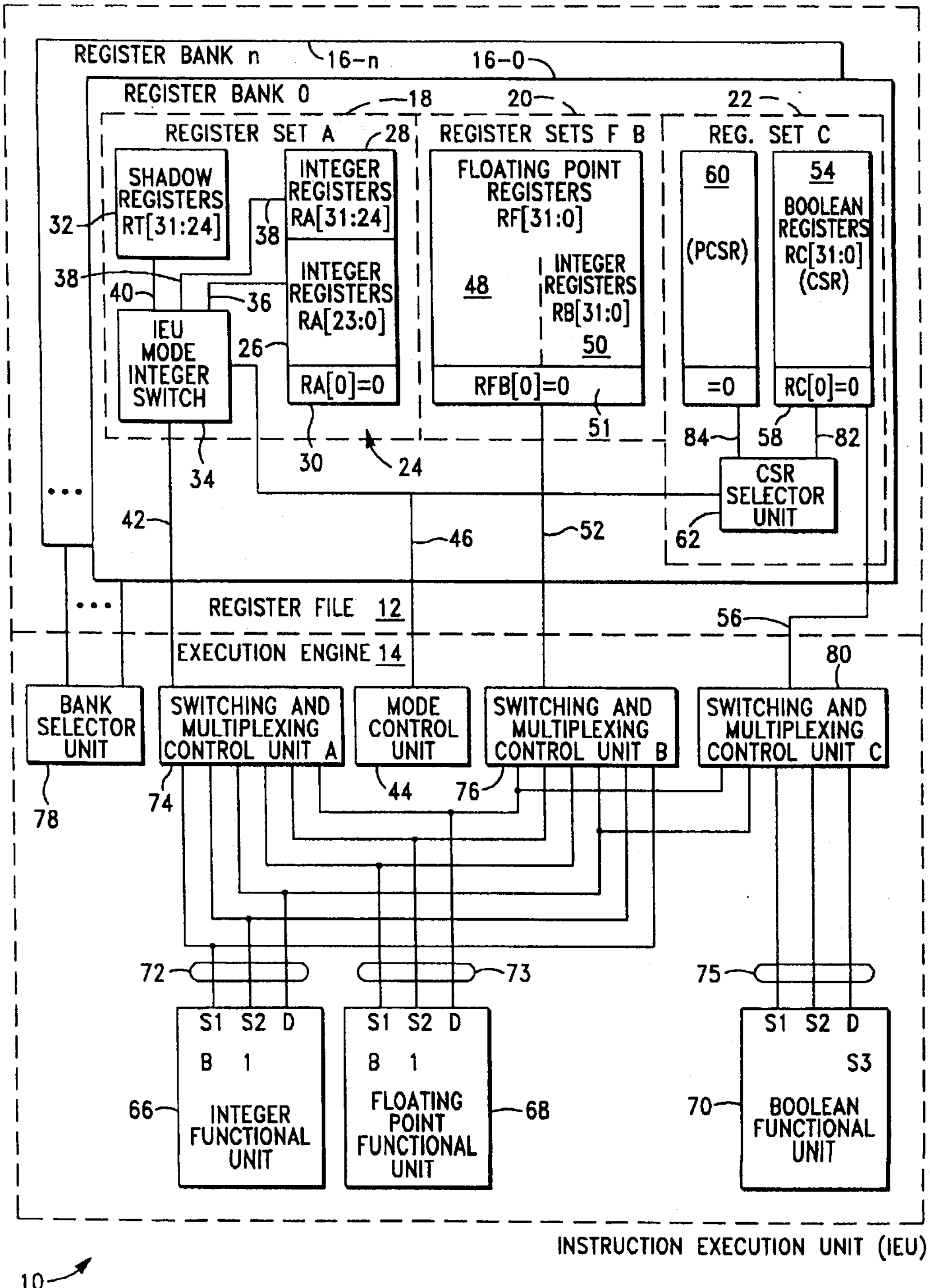
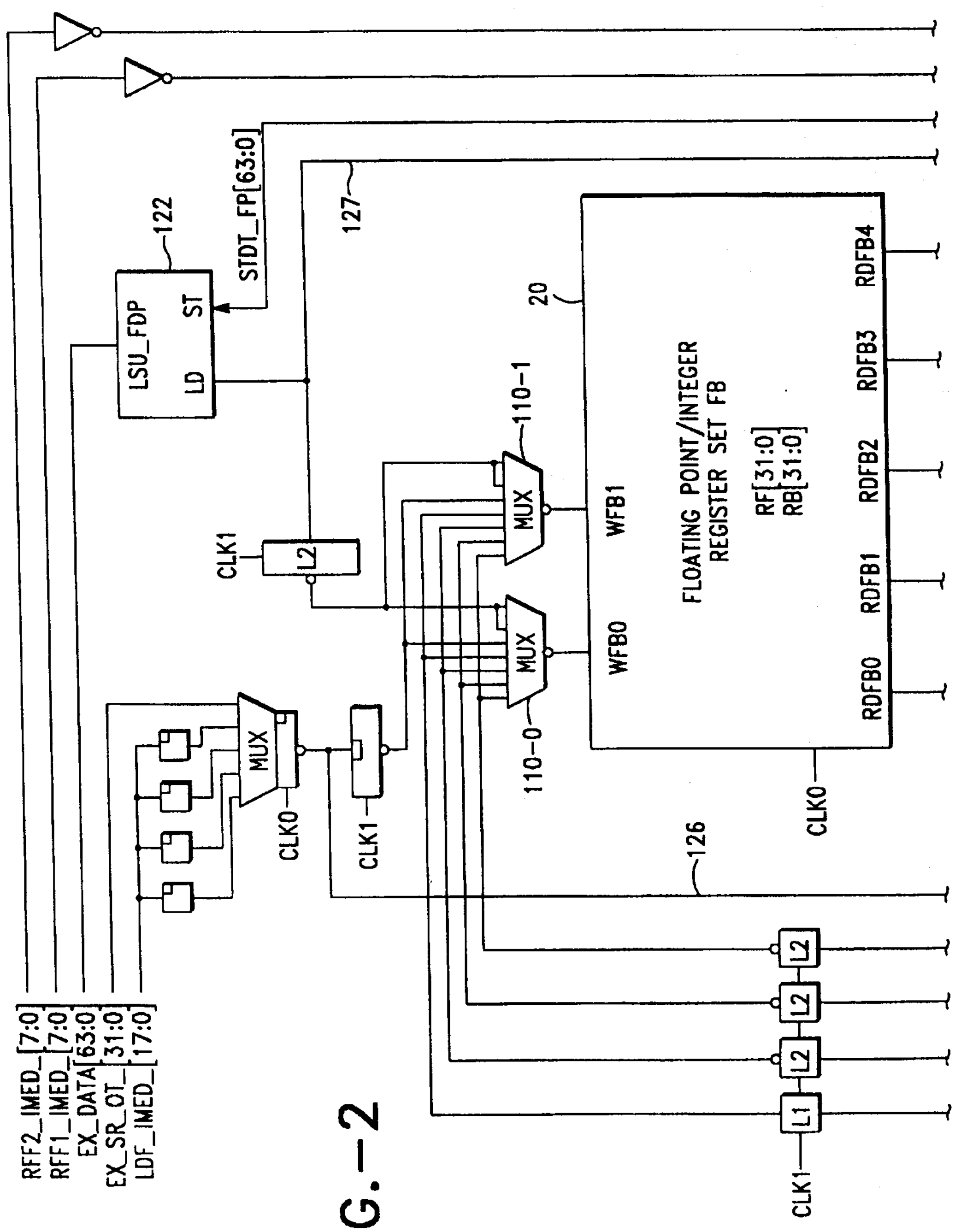

REGISTER BANK n
16-n
16-0
REGISTER BANK 0
18
20
22
REGISTER SET A
28
REGISTER SETS F B
REG. SET C
SHADOW REGISTERS RT[31:24]
INTEGER REGISTERS RA[31:24]
FLOATING POINT REGISTERS RF[31:0]
60
(PCSR)
54
BOOLEAN REGISTERS RC[31:0] (CSR)
32
38
38
40
36
INTEGER REGISTERS RA[23:0]
48
INTEGER REGISTERS RB[31:0]
50
IEU MODE INTEGER SWITCH
26
RA[0]=0
RFB[0]=0
=0
RC[0]=0
51
84
58
82
34
30
24
CSR SELECTOR UNIT
42
46
52
62
REGISTER FILE 12
56
EXECUTION ENGINE 14
80
BANK SELECTOR UNIT
SWITCHING AND MULTIPLEXING CONTROL UNIT A
MODE CONTROL UNIT
SWITCHING AND MULTIPLEXING CONTROL UNIT B
SWITCHING AND MULTIPLEXING CONTROL UNIT C
78
74
44
76
72
73
75
S1 S2 D
B 1
INTEGER FUNCTIONAL UNIT
66
S1 S2 D
B 1
FLOATING POINT FUNCTIONAL UNIT
68
S1 S2 D
S3
BOOLEAN FUNCTIONAL UNIT
70
INSTRUCTION EXECUTION UNIT (IEU)
10

FIG.—1



RFF2_IMED_[7:0]
RFF1_IMED_[7:0]
EX_DATA[63:0]
EX_SR_OT_[31:0]
LDF_IMED_[17:0]
LSU_FDP
LD
ST
122
STDT_FP[63:0]
127
CLK1
L2
MUX
110-1
110-0
20
WFB1
WFB0
FLOATING POINT/INTEGER
REGISTER SET FB
RF[31:0]
RB[31:0]
RDFB4
RDFB3
RDFB2
RDFB1
RDFB0
CLK0
126
L1

FIG.—2

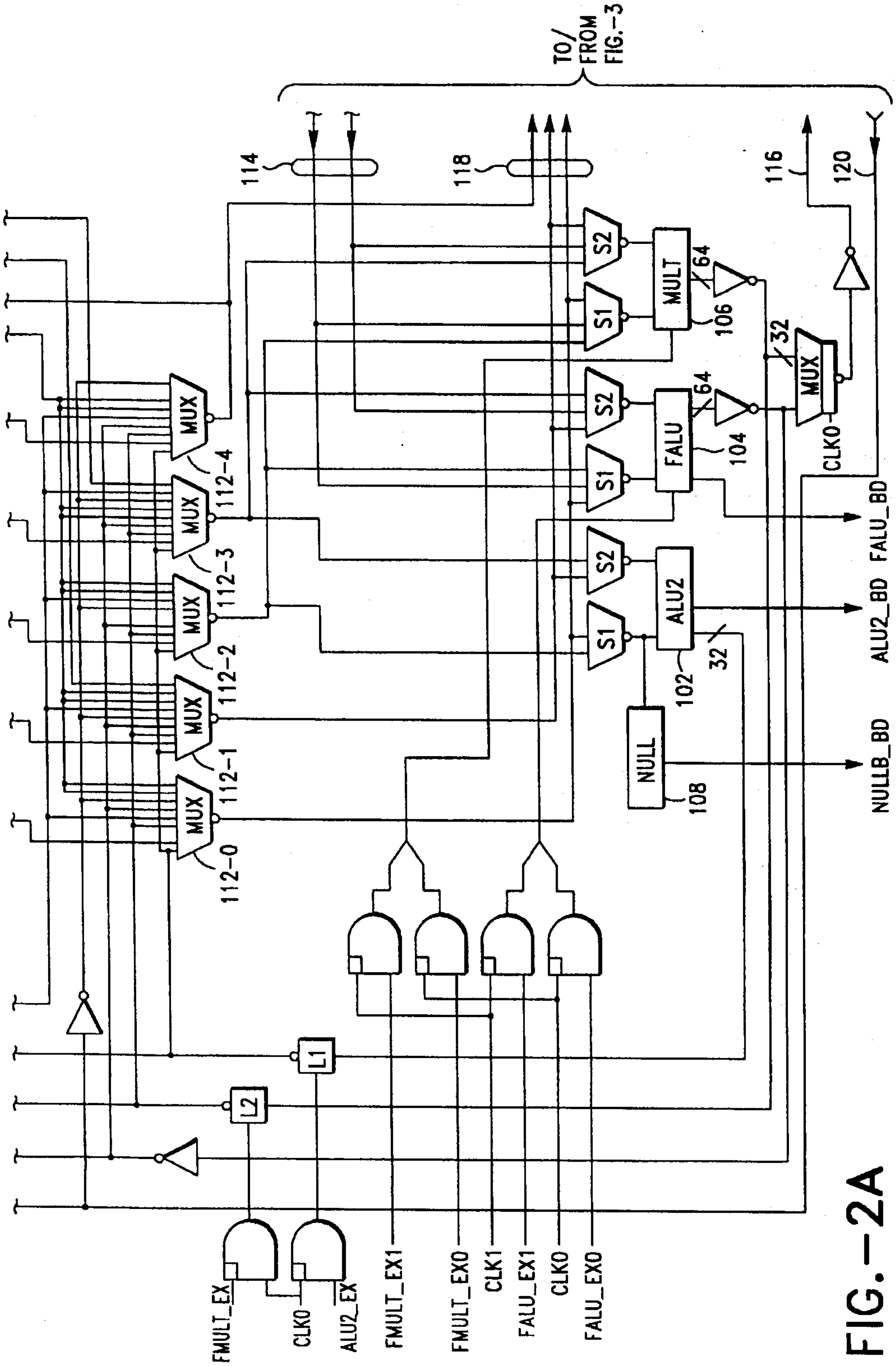

TO/
FROM
FIG.-3
114
118
116
120
S2
S1
MULT
64
106
32
MUX
CLK0
FALU
104
ALU2
102
NULL
108
FALU_BD
ALU2_BD
NULLB_BD
MUX
112-4
112-3
112-2
112-1
112-0
L1
L2
FMULT_EX
CLK0
ALU2_EX
FMULT_EX1
FMULT_EX0
CLK1
FALU_EX1
CLK0
FALU_EX0

FIG.—2A

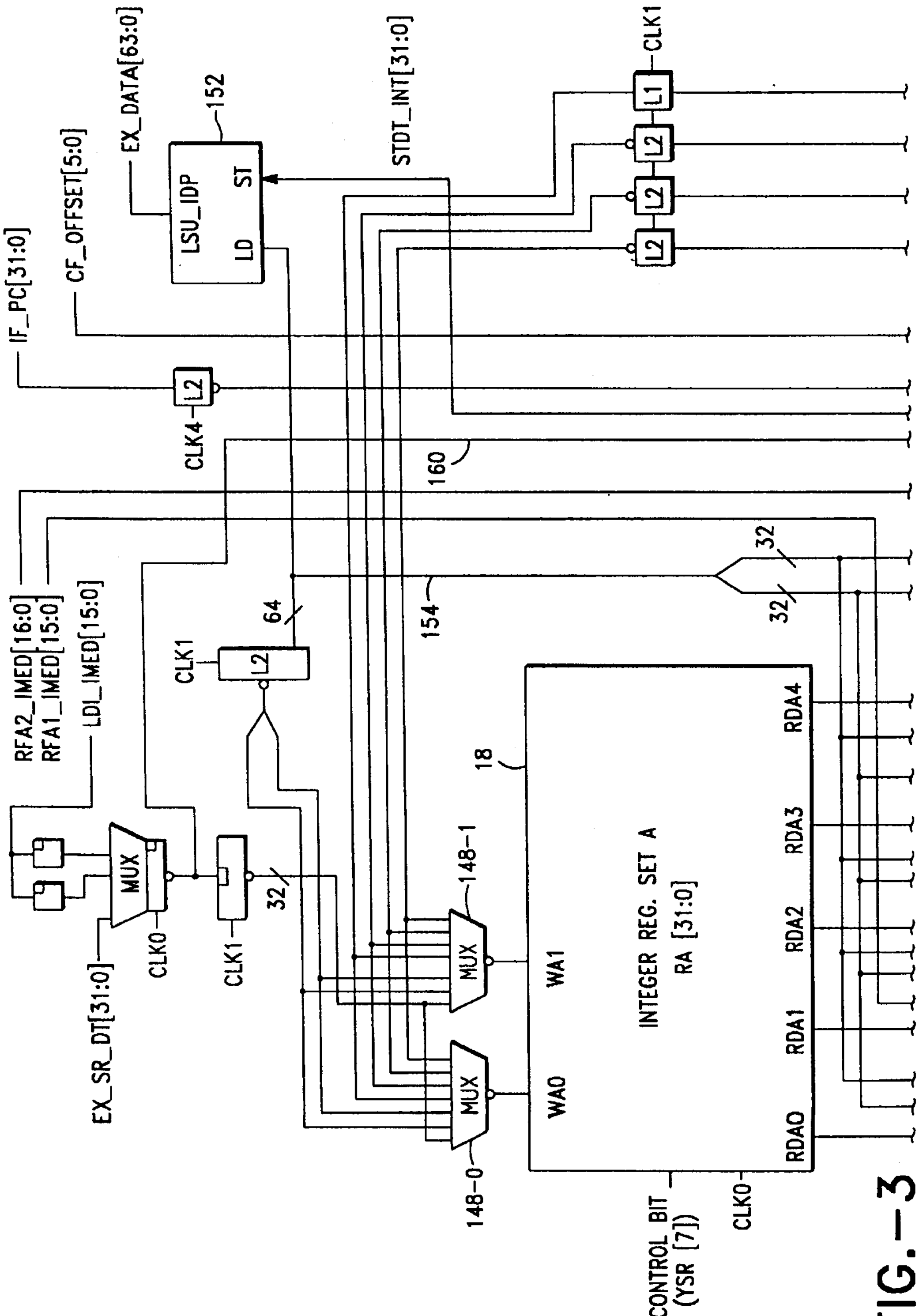

IF_PC[31:0]
CF_OFFSET[5:0]
EX_DATA[63:0]
152
LSU_IDP
LD
ST
STDT_INT[31:0]
CLK1
L1
L2
L2
L2
L2
CLK4
160
RFA2_IMED[16:0]
RFA1_IMED[15:0]
LDI_IMED[15:0]
64
CLK1
L2
154
32
32
18
MUX
CLK0
CLK1
32
EX_SR_DT[31:0]
148-1
MUX
MUX
148-0
WA1
WA0
INTEGER REG. SET A
RA [31:0]
RDA4
RDA3
RDA2
RDA1
RDA0
CONTROL_BIT
(YSR [7])
CLK0

FIG.—3

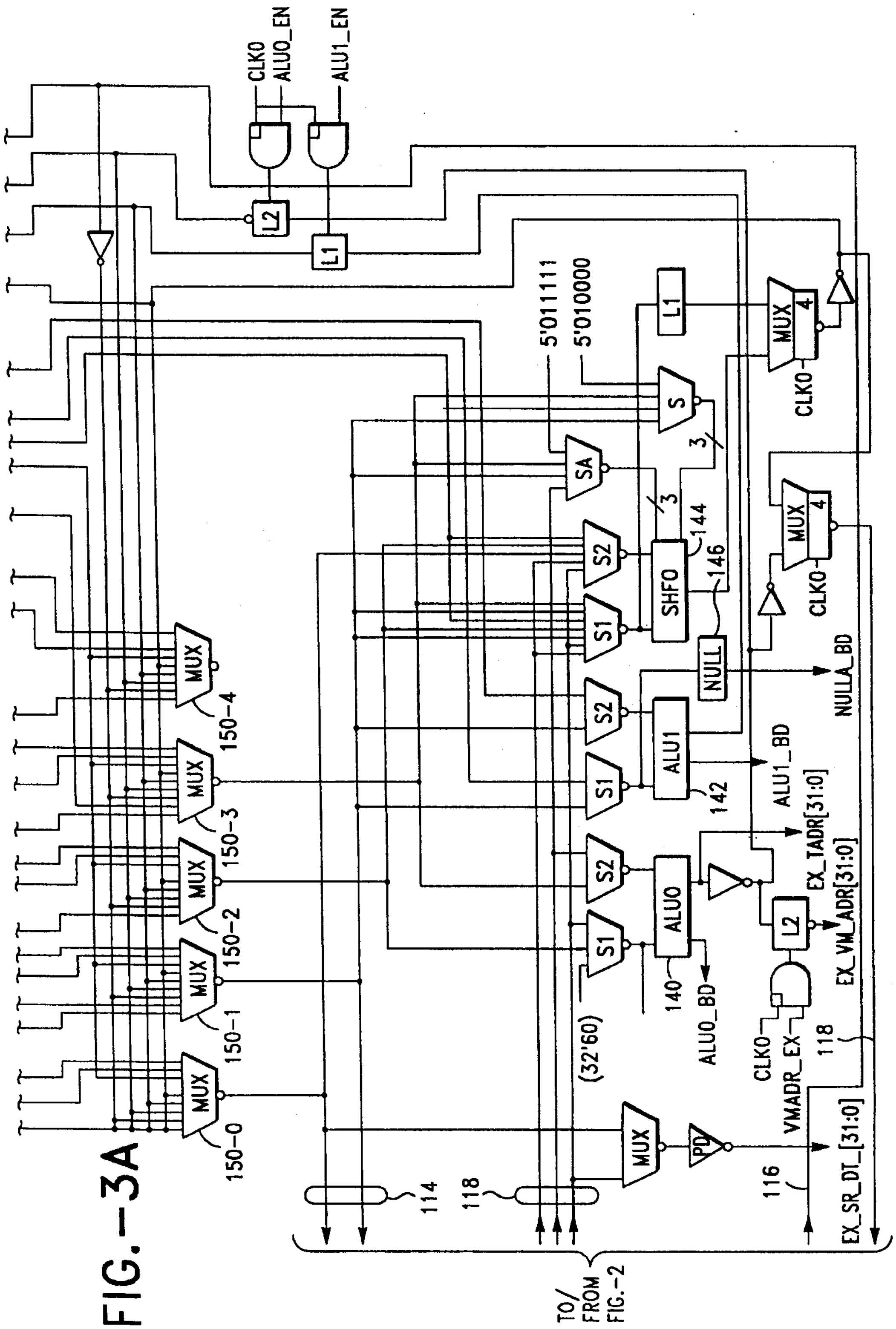

CLK0
ALU0_EN
ALU1_EN
L2
L1
5'011111
5'010000
MUX
4
CLK0
S
3
SA
3
144
S2
SHF0
146
S1
NULL
NULLA_BD
S2
ALU1
S1
142
ALU1_BD
EX_TADR[31:0]
S2
ALU0
L2
EX_VM_ADR[31:0]
S1
140
ALU0_BD
(32'60)
CLK0
VMADR_EX
118
MUX
150-4
150-3
150-2
150-1
150-0
MUX
PD
116
EX_SR_DT_[31:0]
114
118
TO/ FROM FIG.-2

FIG.—3A

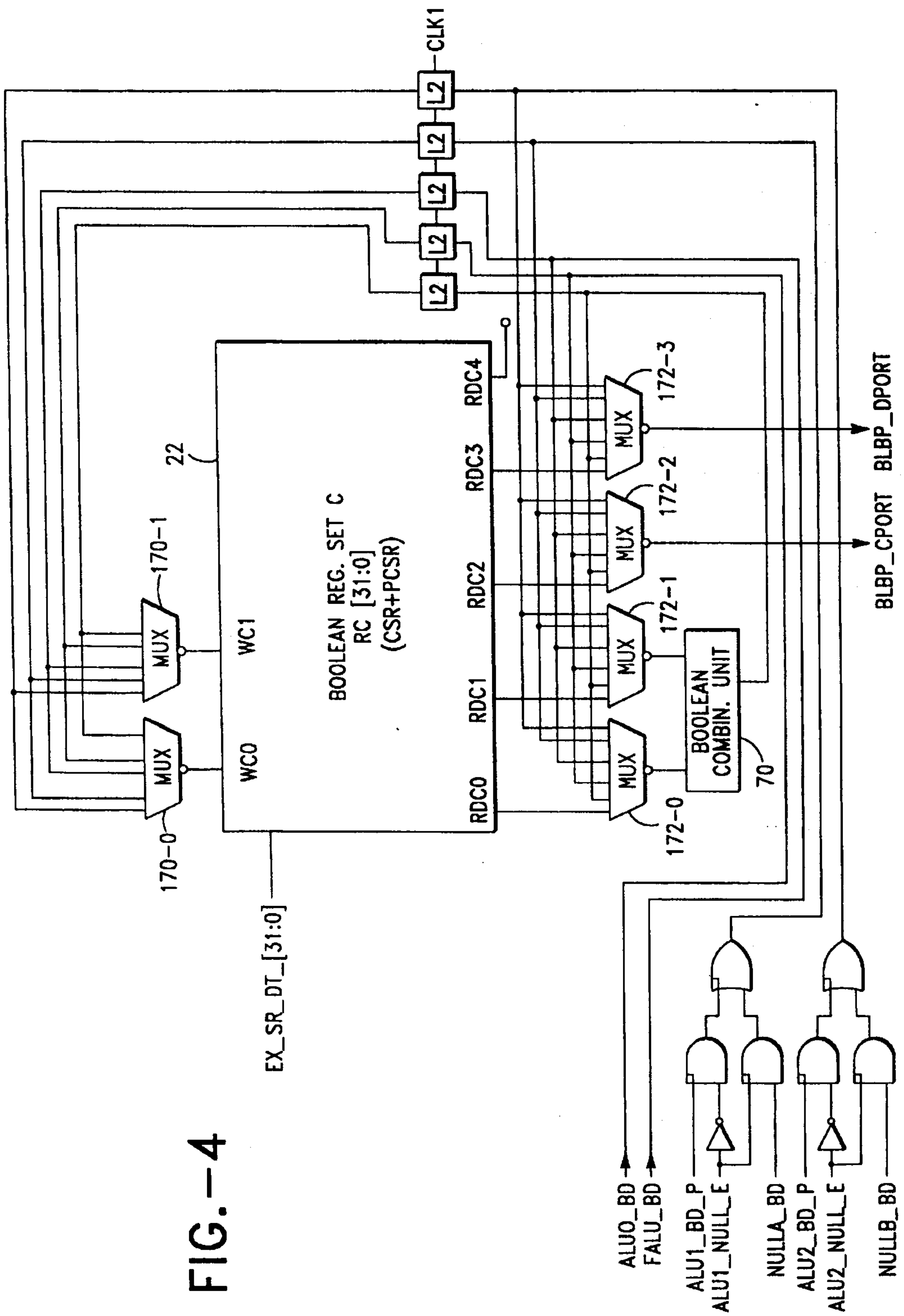

CLK1
L2
L2
L2
L2
L2
22
170-1
170-0
MUX
MUX
WC1
WC0
BOOLEAN REG. SET C
RC [31:0]
(CSR+PCSR)
RDC4
RDC3
RDC2
RDC1
RDC0
172-3
172-2
172-1
172-0
MUX
MUX
MUX
MUX
BOOLEAN
COMBIN. UNIT
70
BLBP_DPORT
BLBP_CPORT
EX_SR_DT_[31:0]
ALU0_BD
FALU_BD
ALU1_BD_P
ALU1_NULL_E
NULLA_BD
ALU2_BD_P
ALU2_NULL_E
NULLB_BD

FIG.-4

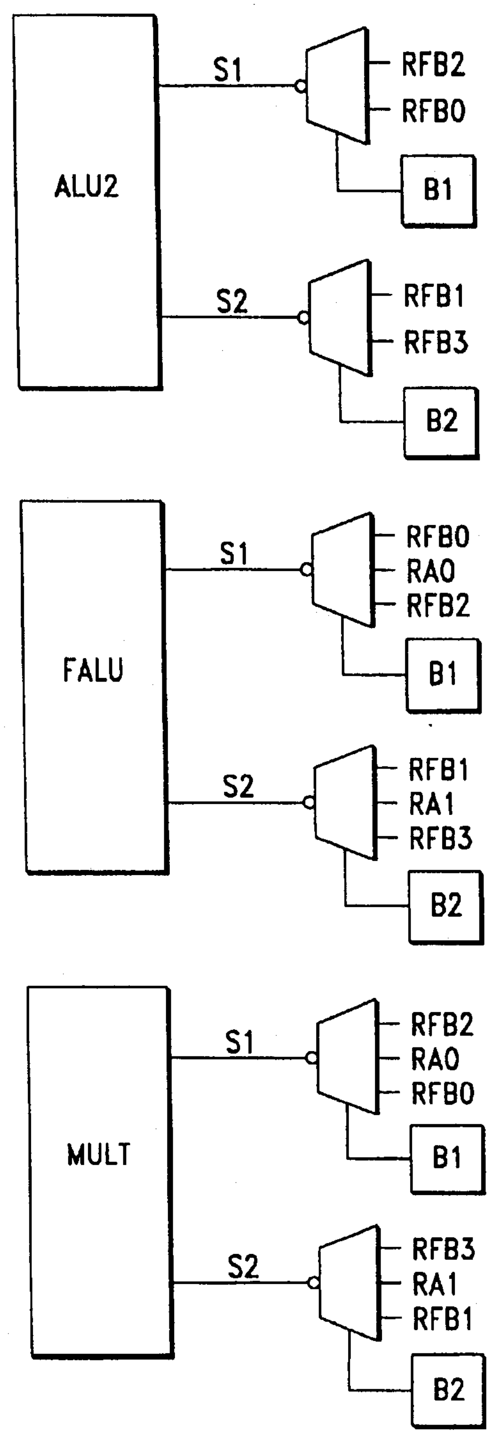

ALU2
S1
RFB2
RFB0
B1
S2
RFB1
RFB3
B2
FALU
S1
RFB0
RA0
RFB2
B1
S2
RFB1
RA1
RFB3
B2
MULT
S1
RFB2
RA0
RFB0
B1
S2
RFB3
RA1
RFB1
B2

FIG.—5

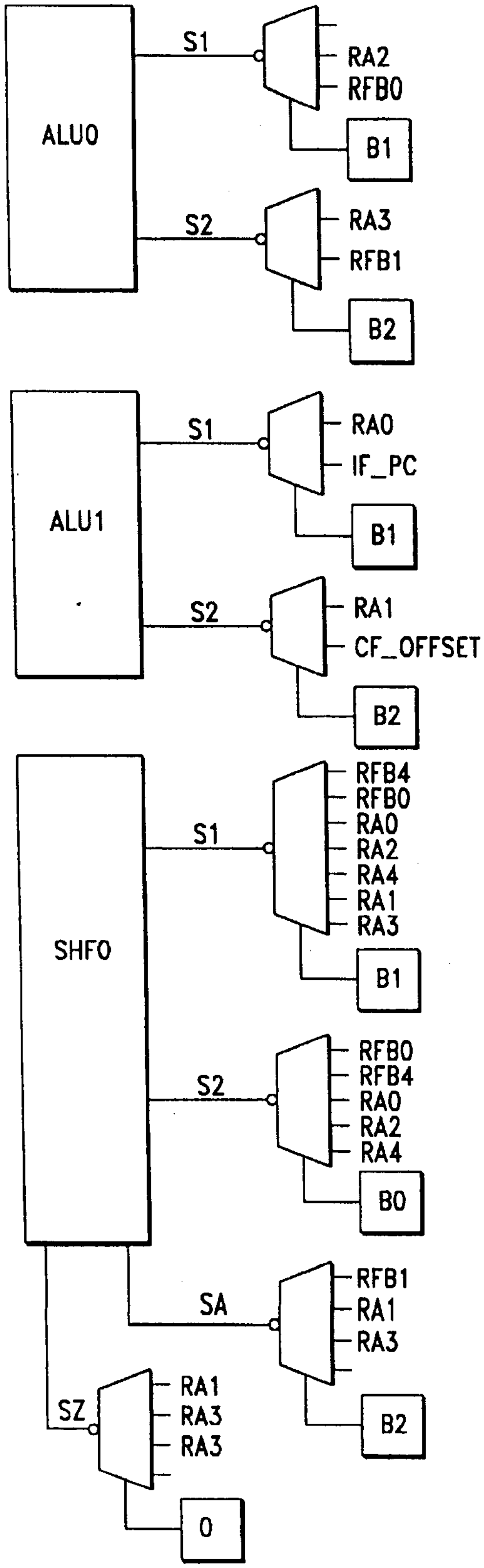
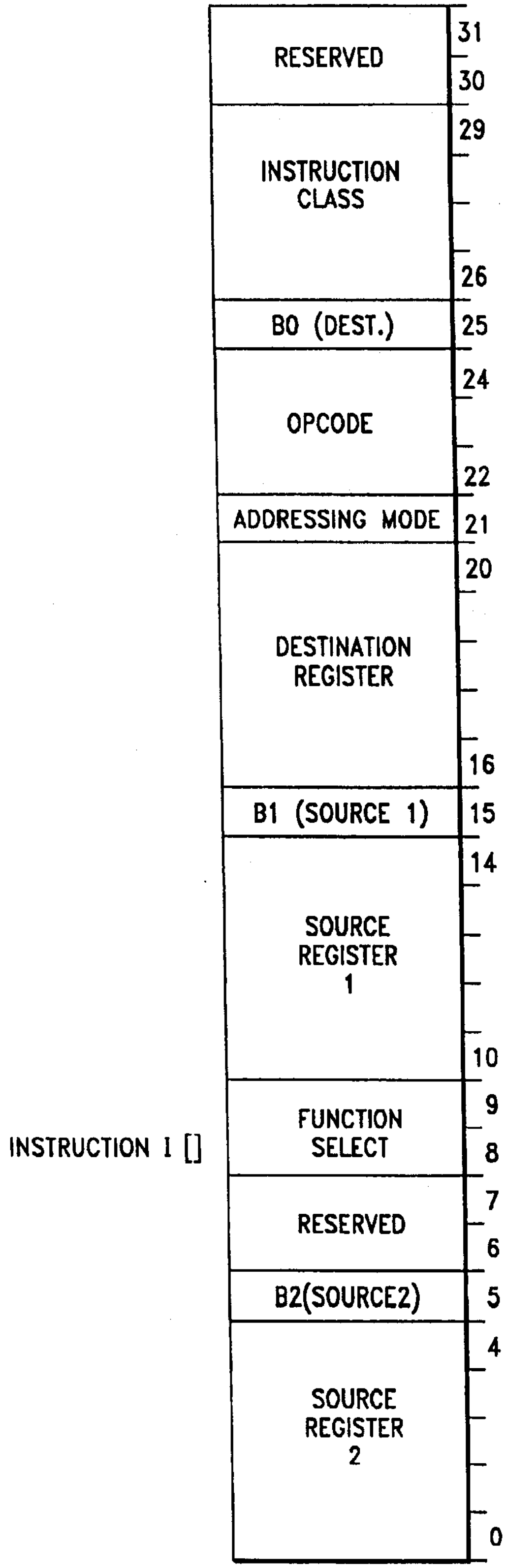

ALU0
S1
RA2
RFB0
B1
S2
RA3
RFB1
B2
ALU1
S1
RA0
IF_PC
B1
S2
RA1
CF_OFFSET
B2
SHF0
S1
RFB4
RFB0
RA0
RA2
RA4
RA1
RA3
B1
S2
RFB0
RFB4
RA0
RA2
RA4
B0
SA
RFB1
RA1
RA3
B2
SZ
RA1
RA3
RA3
0

FIG.—6



RESERVED
31
30
29
INSTRUCTION CLASS
26
B0 (DEST.)
25
24
OPCODE
22
ADDRESSING MODE
21
20
DESTINATION REGISTER
16
B1 (SOURCE 1)
15
14
SOURCE REGISTER 1
10
9
FUNCTION SELECT
INSTRUCTION I []
8
7
RESERVED
6
B2(SOURCE2)
5
4
SOURCE REGISTER 2
0

FIG.—7

RISC MICROPROCESSOR ARCHITECTURE IMPLEMENTING MULTIPLE TYPED REGISTER SETS

The present Application is a Continuation Application of U.S. patent application Ser. No. 08/937,361, filed on Sep. 25, 1997, now U.S. Pat. No. 5,838,986, (allowed), which is a continuation of application Ser. No. 08/665,845, filed on Jun. 19, 1996, now U.S. Pat. No. 5,682,546, (patented), which is a continuation of application Ser. No. 08/465,239, filed Jun. 5, 1995, now U.S. Pat No. 5,560,035, (patented), which is a continuation of application Ser. No. 07/726,773, filed Jul. 8, 1991, now U.S. Pat. No. 5,493,687, (patented).

CROSS-REFERENCE TO RELATED APPLICATION

Applications of particular interest to the present application, include:

1. High-Performance, Superscalar-Based Computer System with Out-of-Order Instruction Execution, application Ser. No. 07/817,810, filed Jan. 8, 1992, now U.S. Pat. No. 5,539,911, by Le Trong Nguyen et al.;
2. High-Performance Superscalar-Based Computer System with Out-of-Order Instruction Execution and Concurrent Results Distribution, application Ser. No. 08/397,016, filed Mar. 1, 1995, now U.S. Pat. No. 5,560,032, by Quang Trang et al.;
3. RISC Microprocessor Architecture with Isolated Architectural Dependencies, application Ser. No. 08/292, 177, filed Aug. 18, 1994, now abandoned, which is a FWC of application Ser. No. 07/817,807, filed Jan. 8, 1992, which is a continuation of application Ser. No. 07/726,744, filed Jul. 8, 1991, by Yoshiyuki Miyayama;
4. RISC Microprocessor Architecture Implementing Fast Trap and Exception State, application Ser. No. 08/345, 333, filed Nov. 21, 1994, now U.S. Pat. No. 5,481,685, by Quang Trang;
5. Page Printer Controller Including a Single Chip Superscalar Microprocessor with Graphics Functional Units, application Ser. No. 08/267,646, filed Jun. 28, 1994, now U.S. Pat. No. 5,394,515, by Derek Lentz et al., and
6. Microprocessor Architecture Capable with a Switch Network for Data Transfer Between Cache, Memory Port, and IOU, application Ser. No. 07/726,893, filed Jul. 8, 1991, now U.S. Pat. No. 5,440,752, by Derek Lentz et al.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates generally to microprocessors, and more specifically to a RISC microprocessor having plural, symmetrical sets of registers.

2. Description of the Background

In addition to the usual complement of main memory storage and secondary permanent storage, a microprocessor-based computer system typically also includes one or more general purpose data registers, one or more address registers, and one or more status flags. Previous systems have included integer registers for holding integer data and floating point registers for holding floating point data. Typically, the status flags are used for indicating certain conditions resulting from the most recently executed operation. There generally are status flags for indicating whether, in the previous operation: a carry occurred, a negative number resulted, and/or a zero resulted.

These flags prove useful in determining the outcome of conditional branching within the flow of program control. For example, if it is desired to compare a first number to a second number and upon the conditions that the two are equal, to branch to a given subroutine, the microprocessor may compare the two numbers by subtracting one from the other, and setting or clearing the appropriate condition flags. The numerical value of the result of the subtraction need not be stored. A conditional branch instruction may then be executed, conditioned upon the status of the zero flag. While being simple to implement, this scheme lacks flexibility and power. Once the comparison has been performed, no further numerical or other operations may be performed before the conditional branch upon the appropriate flag; otherwise, the intervening instructions will overwrite the condition flag values resulting from the comparison, likely causing erroneous branching. The scheme is further complicated by the fact that it may be desirable to form greatly complex tests for branching, rather than the simple equality example given above.

For example, assume that the program should branch to the subroutine only upon the condition that a first number is greater than a second number, and a third number is less than a fourth number, and a fifth number is equal to a sixth number. It would be necessary for previous microprocessors to perform a lengthy series of comparisons heavily interspersed with conditional branches. A particularly undesirable feature of this serial scheme of comparing and branching is observed in any microprocessor having an instruction pipeline.

In a pipelined microprocessor, more than one instruction is being executed at any given time, with the plural instructions being in different stages of execution at any given moment. This provides for vastly improved throughput. A typical pipeline microprocessor may include pipeline stages for: (a) fetching an instruction, (b) decoding the instruction, (c) obtaining the instruction's operands, (d) executing the instruction, and (e) storing the results. The problem arises when a conditional branch instruction is fetched. It may be the case that the conditional branch's condition cannot yet be tested, as the operands may not yet be calculated, if they are to result from operations which are yet in the pipeline. This results in a "pipeline stall", which dramatically slows down the processor.

Another shortcoming of previous microprocessor-based systems is that they have included only a single set of registers of any given data type. In previous architectures, when an increased number of registers has been desired within a given data type, the solution has been simply to increase the size of the single set of those type of registers. This may result in addressing problems, access conflict problems, and symmetry problems.

On a similar note, previous architectures have restricted each given register set to one respective numerical data type. Various prior systems have allowed general purpose registers to hold either numerical data or address "data", but the present application will not use the term "data" to include addresses. What is intended may be best understood with reference to two prior systems. The Intel 8085 microprocessor includes a register pair "HL" which can be used to hold either two bytes of numerical data or one two-byte address. The present application's improvement is not directed to that issue. More on point, the Intel 80486 microprocessor includes a set of general purpose integer data registers and a set of floating point registers, with each set being limited to its respective data type, at least for purposes of direct register usage by arithmetic and logic units.

This proves wasteful of the microprocessor's resources, such as the available silicon area, when the microprocessor is performing operations which do not involve both data types. For example, user applications frequently involve exclusively integer operations, and perform no floating point operations whatsoever. When such a user application is run on a previous microprocessor which includes floating point registers (such as the 80486), those floating point registers remain idle during the entire execution.

Another problem with previous microprocessor register set architecture is observed in context switching or state switching between a user application and a higher access privilege level entity such as the operating system kernel. When control within the microprocessor switches context, mode, or state, the operating system kernel or other entity to which control is passed typically does not operate on the same data which the user application has been operating on. Thus, the data registers typically hold data values which are not useful to the new control entity but which must be maintained until the user application is resumed. The kernel must generally have registers for its own use, but typically has no way of knowing which registers are presently in use by the user application. In order to make space for its own data, the kernel must swap out or otherwise store the contents of a predetermined subset of the registers. This results in considerable loss of processing time to overhead, especially if the kernel makes repeated, short-duration assertions of control.

On a related note, in prior microprocessors, when it is required that a "grand scale" context switch be made, it has been necessary for the microprocessor to expend even greater amounts of processing resources, including a generally large number of processing cycles, to save all data and state information before making the switch. When context is switched back, the same performance penalty has previously been paid, to restore the system to its former state. For example, if a microprocessor is executing two user applications, each of which requires the full complement of registers of each data type, and each of which may be in various stages of condition code setting operations or numerical calculations, each switch from one user application to the other necessarily involves swapping or otherwise saving the contents of every data register and state flag in the system. This obviously involves a great deal of operational overhead, resulting in significant performance degradation, particularly if the main or the secondary storage to which the registers must be saved is significantly slower than the microprocessor itself.

Therefore, we have discovered that it is desirable to have an improved microprocessor architecture which allows the various component conditions of a complex condition to be calculated without any intervening conditional branches. We have further discovered that it is desirable that the plural simple conditions be calculable in parallel, to improve throughput of the microprocessor.

We have also discovered that it is desirable to have an architecture which allows multiple register sets within a given data type.

Additionally, we have discovered it to be desirable for a microprocessor's floating point registers to be usable as integer registers, in case the available integer registers are inadequate to optimally to hold the necessary amount of integer data. Notably, we have discovered that it is desirable that such re-typing be completely transparent to the user application.

We have discovered it to be highly desirable to have a microprocessor which provides a dedicated subset of registers which are reserved for use by the kernel in lieu of at least a subset of the user registers, and that this new set of registers should be addressable in exactly the same manner as the register subset which they replace, in order that the kernel may use the same register addressing scheme as user applications. We have further observed that it is desirable that the switch between the two subsets of registers require no microprocessor overhead cycles, in order to maximally utilize the microprocessor's resources.

Also, we have discovered it to be desirable to have a microprocessor architecture which allows for a "grand scale" context switch to be performed with minimal overhead. In this vein, we have discovered that is desirable to have an architecture which allows for plural banks of register sets of each type, such that two or more user applications may be operating in a multi-tasking environment, or other "simultaneous" mode, with each user application having sole access to at least a full bank of registers. It is our discovery that the register addressing scheme should, desirably, not differ between user applications, nor between register banks, to maximize simplicity of the user applications, and that the system should provide hardware support for switching between the register banks so that the user applications need not be aware of which register bank which they are presently using or even of the existence of other register banks or of other user applications.

These and other advantages of our invention will be appreciated with reference to the following description of our invention, the accompanying drawings, and the claims.

SUMMARY OF THE INVENTION

The present invention provides a register file system comprising: an integer register set including first and second subsets of integer registers, and a shadow subset; a re-typable set of registers which are individually usable as integer registers or as floating point registers; and a set of individually addressable Boolean registers.

The present invention includes integer and floating point functional units which execute integer instructions accessing the integer register set, and which operate in a plurality of modes. In any mode, instructions are granted ordinary access to the first subset of integer registers. In a first mode, instructions are also granted ordinary access to the second subset. However, in a second mode, instructions attempting to access the second subset are instead granted access to the shadow subset, in a manner which is transparent to the instructions. Thus, routines may be written without regard to which mode they will operate in, and system routines (which operate in the second mode) can have at least the second subset seemingly at their disposal, without having to expend the otherwise-required overhead of saving the second subset's contents (which may be in use by user processes operating in the first mode).

The invention further includes a plurality of integer register sets, which are individually addressable as specified by fields in instructions. The register sets include read ports and write ports which are accessed by multiplexers, wherein the multiplexers are controlled by contents of the register set-specifying fields in the instructions.

One of the integer register sets is also usable as a floating point register set. In one embodiment, this set is sixty-four bits wide to hold double-precision floating point data, but only the low order thirty-two bits are used by integer instructions.

The invention includes functional units for performing Boolean operations, and further includes a Boolean register

set for holding results of the Boolean operations such that no dedicated, fixed-location status flags are required. The integer and floating point functional units execute numerical comparison instructions, which specify individual ones of the Boolean registers to hold results of the comparisons. A Boolean functional unit executes Boolean combinational instructions whose sources and destination are specified registers in the Boolean register set. Thus, the present invention may perform conditional branches upon a single result of a complex Boolean function without intervening conditional branch instructions between the fundamental parts of the complex Boolean function, minimizing pipeline disruption in the data processor.

Finally, there are multiple, identical register banks in the system, each bank including the above-described register sets. A bank may be allocated to a given process or routine, such that the instructions within the routine need not specify upon which bank they operate.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a block diagram of the instruction execution unit of the microprocessor of the present invention, showing the elements of the register file.

FIGS. **2**, **2***a*, **3**, **3***a* and **4** are simplified schematic and block diagrams of the floating point, integer and Boolean portions of the instruction execution unit of FIG. **1**, respectively.

FIGS. **5**–**6** are more detailed views of the floating point and integer portions, respectively, showing the means for selecting between register sets.

FIG. **7** illustrates the fields of an exemplary microprocessor instruction word executable by the instruction execution unit of FIG. **1**.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

I. Register File

FIG. **1** illustrates the basic components of the instruction execution unit (IEU) **10** of the RISC (reduced instruction set computing) processor of the present invention. The IEU **10** includes a register file **12** and an execution engine **14**. The register file **12** includes one or more register banks **16-0** to **16**-n. It will be understood that the structure of each register bank **16** is identical to all of the other register banks **16**. Therefore, the present application will describe only register bank **16-0**. The register bank includes a register set A **18**, a register set FB **20**, and a register set C **22**.

In general, the invention may be characterized as a RISC microprocessor having a register file optimally configured for use in the execution of RISC instructions, as opposed to conventional register files which are sufficient for use in the execution of CISC (complex instruction set computing) instructions by CISC processors. By having a specially adapted register file, the execution engine of the microprocessor's IEU achieves greatly improved performance, both in terms of resource utilization and in terms of raw throughput. The general concept is to tune a register set to a RISC instruction, while the specific implementation may involve any of the register sets in the architecture.

A. Register Set A

Register set A **18** includes integer registers **24** (RA[**31**:**0**]), each of which is adapted to hold an integer value datum. In one embodiment, each integer may be thirty-two bits wide. The RA[ ] integer registers **24** include a first plurality **26** of integer registers (RA[**23**:**0**]) and a second plurality **28** of integer registers (RA[**31**:**24**]). The RA[ ] integer registers **24** are each of identical structure, and are each addressable in the same manner, albeit with a unique address within the integer register set **24**. For example, a first integer register **30** (RA[**0**]) is addressable at a zero offset within the integer register set **24**.

RA[**0**] always contains the value zero. It has been observed that user applications and other programs use the constant value zero more than any other constant value. It is, therefore, desirable to have a zero readily available at all times, for clearing, comparing, and other purposes. Another advantage of having a constant, hard-wired value in a given register, regardless of the particular value, is that the given register may be used as the destination of any instruction whose results need not be saved.

Also, this means that the fixed register will never be the cause of a data dependency delay. A data dependency exists when a "slave" instruction requires, for one or more of its operands, to the result of a "master" instruction. In a pipelined processor, this may cause pipeline stalls. For example, the master instruction, although occurring earlier in the code sequence than the slave instruction, may take considerably longer to execute. It will be readily appreciated that if a slave "increment and store" instruction operates on the result data of a master "quadruple-word integer divide" instruction, the slave instruction will be fetched, decoded, and awaiting execution many clock cycles before the master instruction has finished execution. However, in certain instances, the-numerical result of a master instruction is not needed, and the master instruction is executed for some other purpose only, such as to set condition code flags. If the master instruction's destination is RA[**0**], the numerical results will be effectively discarded. The data dependency checker (not shown) of the IEU **10** will not cause the slave instruction to be delayed, as the ultimate result of the master instruction—zero—is already known.

The integer register set A **24** also includes a set of shadow registers **32** (RT[**31**:**24**]). Each shadow register can hold an integer value, and is, in one embodiment, also thirty-two bits wide. Each shadow register is addressable as an offset in the same manner in which each integer register is addressable.

Finally, the register set A includes an IEU mode integer switch **34**. The switch **34**, like other such elements, need not have a physical embodiment as a switch, so long as the corresponding logical functionality is provided within the register sets. The IEU mode integer switch **34** is coupled to the first subset **26** of integer registers on line **36**, to the second subset of integer registers **28** on line **38**, and to the shadow registers **32** on line **40**. All accesses to the register set A **18** are made through the IEU mode integer switch **34** on line **42**. Any access request to read or write a register in the first subset RA[**23**:**0**] is passed automatically through the IEU mode integer switch **34**. However, accesses to an integer register with an offset outside the first subset RA[**23**:**0**] will be directed-either to the second subset RA[**31**:**24**] or the shadow registers RT[**31**;**24**], depending upon the operational mode of the execution engine **14**.

The IEU mode integer switch **34** is responsive to a mode control unit **44** in the execution engine **14**. The mode control unit **44** provides pertinent state or mode information about the IEU **10** to the IEU mode integer switch **34** on line **46**. When the execution engine performs a context switch such as a transfer to kernel mode, the mode control unit **44** controls the IEU mode integer switch **34** such that any requests to the second subset RA[**31**:**24**] are re-directed to the shadow RT[**31**:**24**], using the same requested offset within the integer set. Any operating system kernel or other then-executing entity may thus have apparent access to the

second subset RA[31:24] without the otherwise-required overhead of swapping the contents of the second subset RA[31:24] out to main memory, or pushing the second subset RA[31:24] onto a stack, or other conventional register-saving technique.

When the execution engine **14** returns to normal user mode and control passes to the originally-executing user application, the mode control unit **44** controls the IEU mode integer switch **34** such that access is again directed to the second subset RA[31:24]. In one embodiment, the mode control unit **44** is responsive to the present state of interrupt enablement in the IEU **10**. In one embodiment, the execution engine **14** includes a processor status register (PSR) (not shown), which includes a one-bit flag (PSR[7]) indicating whether interrupts are enabled or disabled. Thus, the line **46** may simply couple the IEU mode integer switch **34** to the interrupts-enabled flag in the PSR. While interrupts are disabled, the IEU **10** maintains access to the integers RA[23:0], in order that it may readily perform analysis of various data of the user application. This may allow improved debugging, error reporting, or system performance analysis.

B. Register Set FB

The re-typable register set FB **20** may be thought of as including floating point registers **48** (RF[31:0]); and/or integer registers **50** (RB[31:0]). When neither data type is implied to the exclusion of the other, this application will use the term RFB[ ]. In one embodiment, the floating point registers RF[ ] occupy the same physical silicon space as the integer registers RB[ ]. In one embodiment, the floating point registers RF[ ] are sixty-four bits wide and the integer registers RB[ ] are thirty-two bits wide. It will be understood that if double-precision floating point numbers are not required, the register set RFB[ ] may advantageously be constructed in a thirty-two-bit width to save the silicon area otherwise required by the extra thirty-two bits of each floating point register.

Each individual register in the register set RFB[ ] may hold either a floating point value or an integer value. The register set RFB[ ] may include optional hardware for preventing accidental access of a floating point value as though it were an integer value, and vice versa. In one embodiment, however, in the interest of simplifying the register set RFB[ ], it is simply left to the software designer to ensure that no erroneous usages of individual registers are made. Thus, the execution engine **14** simply makes an access request on line **52**, specifying an offset into the register set RFB[ ], without specifying whether the register at the given offset is intended to be used as a floating point register or an integer register. Within the execution engine **14**, various entities may use either the full sixty-four bits provided by the register set RFB[ ], or may use only the low order thirty-two bits, such as in integer operations or single-precision floating point operations.

A first register RFB[0] **51** contains the constant value zero, in a form such that RB[0] is a thirty-two-bit integer zero (0000_{hex}) and RF[0] is a sixty-four-bit floating point zero (00000000_{hex}) This provides the same advantages as described above for RA[0].

C. Register Set C

The register set C **22** includes a plurality of Boolean registers **54** (RC[31:0]). RC[ ] is also known as the "condition status register" (CSR). The Boolean registers RC[ ] are each identical in structure and addressing, albeit that each is individually addressable at a unique address or offset within RC[ ].

In one embodiment, register set C further includes a "previous condition status register" (PCSR) **60**, and the register set C also includes a CSR selector unit **62**, which is responsive to the mode control unit **44** to select alternatively between the CSR **54** and the PCSR **60**. In the one embodiment, the CSR is used when interrupts are enabled, and the PCSR is used when interrupts are disabled. The CSR and PCSR are identical in all other respects. In the one embodiment, when interrupts are set to be disabled, the CSR selector unit **62** pushes the contents of the CSR into the PCSR, overwriting the former contents of the PCSR, and when interrupts are re-enabled, the CSR selector unit **62** pops the contents of the PCSR back into the CSR. In other embodiments it may be desirable to merely alternate access between the CSR and the PCSR, as is done with RA[31:24] and RT[31:24]. In any event, the PCSR is always available as a thirty-two-bit "special register".

None of the Boolean registers is a dedicated condition flag, unlike the Boolean registers in previously known microprocessors. That is, the CSR **54** does not include a dedicated carry flag, nor a dedicated a minus flag, nor a dedicated flag indicating equality of a comparison or a zero subtraction result. Rather, any Boolean register may be the destination of the Boolean result of any Boolean operation. As with the other register sets, a first Boolean register **58** (RC[0]) always contains the value zero, to obtain the advantages explained above for RA[0]. In the preferred embodiment, each Boolean register is one bit wide, indicating one Boolean value.

II. Execution Engine

The execution engine **14** includes one or more integer functional units **66**, one or more floating point functional units **68**, and one or more Boolean functional units **70**. The functional units execute instructions as will be explained below. Buses **72, 73**, and **75** connect the various elements of the IEU **10**, and will each be understood to represent data, address, and control paths.

A. Instruction Format

FIG. **7** illustrates one exemplary format for an integer instruction which the execution engine **14** may execute. It will be understood that not all instructions need to adhere strictly to the illustrated format, and that the data processing system includes an instruction fetcher and decoder (not shown) which are adapted to operate upon varying format instructions. The single example of FIG. **7** is for ease in explanation only. Throughout this application the identification I[ ] will be used to identify various bits of the instruction. I[31:30] are reserved for future implementations of the execution engine **14**. I[29:26] identify the instruction class of the particular instruction. Table 1 shows the various classes of instructions performed by the present invention.

TABLE 1

Instruction Classes

Class	Instructions
0–3	Integer and floating point register-to-register instructions
4	Immediate constant load
5	Reserved
6	Load
7	Store
8–11	Control Flow
12	Modifier
13	Boolean operations
14	Reserved
15	Atomic (extended)

Instruction classes of particular interest to this Application include the Class **0-3** register-to-register instructions and the

Class 13 Boolean operations. While other classes of instructions also operate upon the register file 12, further discussion of those classes is not believed necessary in order to fully understand the present invention.

I[25] is identified as B0, and indicates whether the destination register is in register set A or register set B. I[24:22] are an opcode which identifies, within the given instruction class, which specific function is to be performed. For example, within the register-to-register classes an opcode may specify "addition". I[21] identifies the addressing mode which is to be used when performing the instruction—either register source addressing or immediate source addressing. I[20:16] identify the destination register as an offset within the register set indicated by B0. I[15] is identified as B1 and indicates whether the first operand is to be taken from register set A or register set B. I[14:10] identify the register offset from which the first operand is to be taken. I[9:8] identify a function selection—an extension of the opcode I[24:22]. I[7:6] are reserved. I[5] is identified as B2 and indicates whether a second operand of the instruction is to be taken from register set A or register set B. Finally, I[4:0] identify the register offset from which the second operand is to be taken.

With reference to FIG. 1, the integer functional unit 66 and floating point functional unit 68 are equipped to perform integer comparison instructions and floating point comparisons, respectively. The instruction format for the comparison instruction is substantially identical to that shown in FIG. 7, with the caveat that various fields may advantageously be identified by slightly different names. I[20:16] identifies the destination register where the result is to be stored, but the addressing mode field I[21] does not select between register sets A or B. Rather, the addressing mode field indicates whether the second source of the comparison is found in a register or is immediate data. Because the comparison is a Boolean type instruction, the destination register is always found in register set C. All other fields function as shown in FIG. 7. In performing Boolean operations within the integer and floating point functional units, the opcode and function select fields identify which Boolean condition is to be tested for in comparing the two operands. The integer and the floating point functional units fully support the IEEE standards for numerical comparisons.

The IEU 10 is a load/store machine. This means that when the contents of a register are stored to memory or read from memory, an address calculation must be performed in order to determine which location in memory is to be the source or the destination of the store or load, respectively. When this is the case, the destination register field I[20:16] identifies the register which is the destination or the source of the load or store, respectively. The source register 1 field, I[14:10], identifies a register in either set A or B which contains a base address of the memory location.. In one embodiment, the source register 2 field, I[4:0], identifies a register in set A or set B which contains an index or an offset from the base. The load/store address is calculated by adding the index to the base. In another mode, I[7:0] include immediate data which are to be added as an index to the base.

B. Operation of the Instruction Execution Unit and Register Sets

It will be understood by those skilled in the art that the integer functional unit 66, the floating point functional unit 68, and the Boolean functional unit 70 are responsive to the contents of the instruction class field, the opcode field, and the function select field of a present instruction being executed.

1. Integer Operations

For example, when the instruction class, the opcode, and function select indicate that an integer register-to-register addition is to be performed, the integer functional unit may be responsive thereto to perform the indicated operation, while the floating point functional unit and the Boolean functional unit may be responsive thereto to not perform the operation. As will be understood from the cross-referenced applications, however, the floating point functional unit 68 is equipped to perform both floating point and integer operations. Also, the functional units are constructed to each perform more than one instruction simultaneously.

The integer functional unit 66 performs integer functions only. Integer operations typically involve a first source, a second source, and a destination. A given integer instruction will specify a particular operation to be performed on one or more source operands and will specify that the result of the integer operation is to be stored at a given destination. In some instructions, such as address calculations employed in load/store operations, the sources are utilized as a base and an index. The integer functional unit 66 is coupled to a first bus 72 over which the integer functional unit 66 is connected to a switching and multiplexing control (SMC) unit A 74 and an SMC unit B 76. Each integer instruction executed by the integer functional unit 66 will specify whether each of its: sources and destination reside in register set A or register set B.

Suppose that the IEU 10 has received, from the instruction fetch unit (not shown), an instruction to perform an integer register-to-register addition. In various embodiments, the instruction may specify a register bank, perhaps even a separate bank for each source and destination. In one embodiment, the instruction I[ ] is limited to a thirty-two-bit length, and does not contain any indication of which register bank 16-0 through 16-n is involved in the instruction. Rather, the bank selector unit 78 controls which register bank is presently active. In one embodiment, the bank selector unit 78 is responsive to one or more bank selection bits in a status word (not shown) within the IEU 10.

In order to perform the integer addition instruction, the integer functional unit 66 is responsive to the identification in I[14:10] and I[4:0] of the first and second source registers. The integer functional unit 66 places an identification of the first and second source registers at ports S1 and S2, respectively, onto the integer functional unit bus 72 which is coupled to both SMC units A and B 74 and 76. In one embodiment, the SMC units A and B are each coupled to receive B0-2 from the instruction I[ ]. In one embodiment, a zero in any respective Bn indicates register set A, and a one indicates register set B. During load/store operations, the source ports of the integer and floating point functional units 66 and 68 are utilized as a base port and an index port, B and I, respectively.

After obtaining the first and second operands from the indicated register sets on the bus 72, as explained below, the integer functional unit 66 performs the indicated operation upon those operands, and provides the result at port D onto the integer functional unit bus 72. The SMC units A and B are responsive to B0 to route the result to the appropriate register set A or B.

The SMC unit B is further responsive to the instruction class, opcode, and function selection to control whether operands are read from (or results are stored to) either a floating point register RF[ ] or an integer register RB[ ]. As indicated, in one embodiment, the registers RF[ ] may be sixty-four bits wide while the registers are RB[ ] are only thirty-two bits wide. Thus, SMC unit B controls whether a

word or a double word is written to the register set RFB[ ]. Because all registers within register set A are thirty-two bits wide, SMC unit A need not include means for controlling the width of data transfer on the bus **42**.

All data on the bus **42** are thirty-two bits wide, but other sorts of complexities exist within register set A. The IEU mode integer switch **34** is responsive to the mode control unit **44** of the execution engine **14** to control whether data on the bus **42** are connected through to bus **36**, bus **38** or bus **40**, and vice versa.

IEU mode integer switch **34** is further responsive to I[**20**:**16**], I[**14**:10], and I[**4**:**0**]. If a given indicated destination or source is in RA[**23**:**0**], the IEU mode integer switch **34** automatically couples the data between lines **42** and **36**. However, for registers RA[**31**:**24**], the IEU mode integer switch **34** determines whether data on line **42** is connected to line **38** or line **40**, and vice versa. When interrupts are enabled, IEU mode integer switch **34** connects the SMC unit A to the second subset **28** of integer registers RA[**31**:**24**]. When interrupts are disabled, the IEU mode integer switch **34** connects the SMC unit A to the shadow registers RT[**31**:**24**]. Thus, an instruction executing within the integer functional unit **66** need not be concerned with whether to address RA[**31**:**24**] or RT[**31**:**24**]. It will be understood that SMC unit A may advantageously operate identically whether it is being accessed by the integer functional unit **66** or by the floating point functional unit **68**.

2. Floating Point Operations

The floating point functional unit **68** is responsive to the class, opcode, and function select fields of the instruction, to perform floating point operations. The **S1**, **S2**, and D ports operate as described for the integer functional unit **66**. SMC unit B is responsive to retrieve floating point operands from, and to write numerical floating point results to, the floating point registers RF[ ] on bus **52**.

3. Boolean Operations

SMC unit C-**80** is responsive to the instruction class, opcode, and function select fields of the instruction I[ ]. When SMC unit C detects that a comparison operation has been performed by one of the numerical functional units **66** or **68**, it writes the Boolean result over bus **56** to the Boolean register indicated at the D port of the functional unit which performed the comparison.

The Boolean functional unit **70** does not perform comparison instructions as do the integer and floating point functional units **66** and **68**. Rather, the Boolean functional unit **70** is only used in performing bitwise logical combination of Boolean registers contents, according to the Boolean functions listed in Table 2.

TABLE 2

Boolean Functions	
I[23,22,9,8]	Boolean result calculation
0000	ZERO
0001	S1 AND S2
0010	S1 AND (NOT S2)
0011	S1
0100	(NOT S1) AND S2
0101	S2
0110	S1 XOR S2
0111	S1 OR S2
1000	S1 NOR S2
1001	S1 XNOR S2
1010	NOT S2
1011	S1 OR (NOT S2)
1100	NOT S1

TABLE 2-continued

Boolean Functions	
I[23,22,9,8]	Boolean result calculation
1101	(NOT S1) OR S2
1110	S1 NAND S2
1111	ONE

The advantage which the present invention obtains by having a plurality of homogenous Boolean registers, each of which is individually addressable as the destination of a Boolean operation, will be explained with reference to Tables 3–5. Table 3 illustrates an example of a segment of code which performs a conditional branch based upon a complex Boolean function. The complex Boolean function includes three portions which are OR-ed together. The first portion includes two sub-portions, which are AND-ed together.

TABLE 3

Example of Complex Boolean Function	
1	RA[1] := 0;
2	IF (((RA[2] = RA[3]) AND (RA[4] > RA[5])) OR
3	(RA[6] < RA[7]) OR
4	(RA[8] <> RA[9])) THEN
5	X( )
6	ELSE
7	Y( );
8	RA[10] := 1;

Table 4 illustrates, in pseudo-assembly form, one likely method by which previous microprocessors would perform the function of Table 3. The code in Table 4 is written as though it were constructed by a compiler of at least normal intelligence operating upon the code of Table 3. That is, the compiler will recognize that the condition expressed in lines 2–4 of Table 3 is passed if any of the three portions is true.

TABLE 4

Execution of Complex Boolean Function Without Boolean Register Set			
1	START	LDI	RA[1],0
2	TEST1	CMP	RA[2],RA[3]
3		BNE	TEST2
4		CMP	RA[4],RA[5]
5		BGT	DO_IF
6	TEST2	CMP	RA[6],RA[7]
7		BLT	DO_IF
8	TEST3	CMP	RA[8],RA[9]
9		BEQ	DO_ELSE
10	DO_IF	JSR	ADDRESS OF X( )
11		JMP	PAST_ELSE
12	DO_ELSE	JSR	ADDRESS OF Y( )
13	PAST_ELSE	LDI	RA[10],1

The assignment at line 1 of Table 3 is performed by the "load immediate" statement at line 1 of Table 4. The first portion of the complex Boolean condition, expressed at line 2 of Table 3, is represented by the statements in lines 2–5 of Table 4. To test whether RA[**2**] equals RA[**3**], the compare statement at line 2 of Table 4 performs a subtraction of RA[**2**] from RA[**3**] or vice versa, depending upon the implementation, and may or may not store the result of that subtraction. The important function performed by the comparison statement is that the zero, minus, and carry flags will be appropriately set or cleared.

The conditional branch statement at line 3 of Table 4 branches to a subsequent portion of code upon the condition that RA[2] did not equal RA[3]. If the two were unequal, the zero flag will be clear, and there is no need to perform the second sub-portion. The existence of the conditional branch statement at line 3 of Table 4 prevents the further fetching, decoding, and executing of any subsequent statement in Table 4 until the results of the comparison in line 2 are known, causing,a pipeline stall. If the first sub-portion of the first portion (TEST1) is passed, the second sub-portion at line 4 of Table 4 then compares RA[4] to RA[5], again setting and clearing the appropriate status flags.

If RA[2] equals RA[3], and RA[4] is greater than RA[5], there is no need to test the remaining two portions (TEST2 and TEST3) in the complex Boolean function, and the statement at Table 4, line 5, will conditionally branch to the label DO_IF, to perform the operation inside the "IF" of Table 3. However, if the first portion of the test is failed, additional processing is required to determine which of the "IF" and "ELSE" portions should be executed.

The second portion of the Boolean function is the comparison of RA[6] to RA[7], at line 6 of Table 4, which again sets and clears the appropriate status flags. If the condition "less than" is indicated by the status flags, the complex Boolean function is passed, and execution may immediately branch to the DO_IF label. In various prior microprocessors, the "less than" condition may be tested by examining the minus flag. If RA[7] was not less than RA[6], the third portion of the test must be performed. The statement at line 8 of Table 4 compares RA[8] to RA[9]. If this comparison is failed, the "ELSE" code should be executed; otherwise, execution may simply fall through to the "IF" code at line 10 of Table 4, which is followed by an additional jump around the "ELSE" code. Each of the conditional branches in Table 4, at lines 3, 5, 7 and 9, results in a separate pipeline stall, significantly increasing the processing time required for handling this complex Boolean function.

The greatly improved throughput which results from employing the Boolean register set C of the present invention will now readily be seen with specific reference to Table 5.

TABLE 5

Execution of Complex Boolean Function With Boolean Register Set

1	START	LDI	RA[1],0
2	TEST1	CMP	RC[11],RA[2],RA[3],EQ
3		CMP	RC[12],RA[4],RA[5],GT
4	TEST2	CMP	RC[13],RA[6],RA[7],LT
5	TEST3	CMP	RC[14],RA[8],RA[9],NE
6	COMPLEX	AND	RC[15],RC[11],RC[12]
7		OR	RC[16],RC[13],RC[14]
8		OR	RC[17],RC[15],RC[16]
9		BC	RC[17],DO_ELSE
10	DO_IF	JSR	ADDRESS OF X( )
11		JMP	PAST_ELSE
12	DO_ELSE	JSR	ADDRESS OF Y( )
13	PAST_ELSE	LDI	RA[10],1

Most notably seen at lines 2–5 of Table 5, the Boolean register set C allows the microprocessor to perform the three test portions back-to-back without intervening branching. Each Boolean comparison specifies two operands, a destination, and a Boolean condition for which to test. For example, the comparison at line 2 of Table 5 compares the contents of RA[2] to the contents of RA[3], tests them for equality, and stores into RC[11] the Boolean value of the result of the comparison. Note that each comparison of the Boolean function stores its respective intermediate results in a separate Boolean register. As will be understood with reference to the above-referenced related applications, the IEU **10** is capable of simultaneously performing more than one of the comparisons.

After at least the first two comparisons at lines 2–3 of Table 5 have been completed, the two respective comparison results are AND-ed together as shown at line 6 of Table 3. RC[15] then holds the result of the first portion of the test. The results of the second and third sub-portions of the Boolean function are OR-ed together as seen in Table 5, line 7. It will be understood that, because there are no data dependencies involved, the AND at line 6 and the OR-ed in line 7 may be performed in parallel. Finally, the results of those two operations are OR-ed together as seen at line 8 of Table 5. It will be understood that register RC[17] will then contain a Boolean value indicating the truth or falsity of the entire complex Boolean function of Table 3. It is then possible to perform a single conditional branch, shown at line 9 of Table 5. In the mode shown in Table 5, the method branches to the "ELSE" code if Boolean register RC[17] is clear, indicating that the complex function was failed. The remainder of the code may be the same as it was without the Boolean register set as seen in Table 4.

The Boolean functional unit **70** is responsive to the instruction class, opcode, and function select fields as are the other functional units. Thus, it will be understood with reference to Table 5 again, that the integer and/or floating point functional units will perform the instructions in lines 1–5 and 13, and the Boolean functional unit **70** will perform the Boolean bitwise combination instructions in lines 6–8. The control flow and branching instructions in line 9–12 will be performed by elements of the IEU **10** which are not shown in FIG. **1**.

III. Data Paths

FIGS. **2–5** illustrate further details of the data paths within the floating point, integer, and Boolean portions of the IEU, respectively.

A. Floating Point Portion Data Paths

As seen in FIG. **2**, the register set FB **20** is a multi-ported register set. In one embodiment, the register set FB **20** has two write ports **WFB0-1**, and five read ports **RDFB0-4**. The-floating point functional unit **68** of FIG. **1** is comprised of the **ALU2 102**, **FALU 104**, **MULT 106**, and **NULL 108** of FIG. **2**. All elements of FIG. **2** except the register set **20** and the elements **102–108** comprise the SMC unit B of FIG. **1**.

External, bidirectional data bus EX_DATA[ ] provides data to the floating point load/store unit **122**. Immediate floating point data bus LDF_IMED[ ] provides data from a "load immediate" instruction. Other immediate floating point data are provided on busses RFF1_IMED and RFF2_IMED, such as is involved in an "add immediate" instruction. Data are also provided on bus EX_SR_DT[ ], in response to a "special register move" instruction. Data may also arrive from the integer portion, shown in FIG. **3**, on busses **114** and **120**.

The floating point register set's two write ports **WFB0** and **WFB1** are coupled to write multiplexers **110-0** and **110-1**, respectively. The write multiplexers **110** receive data from: the **ALU0** or **SHF0** of the integer portion of FIG. **3**; the FALU; the MULT; the ALU2; either EX_SR DT[ ] or LDF_IMED[ ]; and EX_DATA[ ]. Those skilled in the art will understand that control signals (not shown) determine which input is selected at each port, and address signals (not shown) determine to which register the input data are written. Multiplexer control and register addressing are within the skill of persons in the art, and will not be discussed for any multiplexer or register set in the present invention.

The floating point register set's five read ports RDFB0 to RDFB4 are coupled to read multiplexers **112-0** to **112-4**, respectively. The read multiplexers each also receives data from: either EX_SR_DT[ ] or LDF_IMED[ ], on load immediate bypass bus **126**; a load external data bypass bus **127**, which allows external load data to skip the register set FB; the output of the ALU2 **102**, which performs non-multiplication integer operations; the FALU **104**, which performs non-multiplication floating point operations; the MULT **106**, which performs multiplication operations; and either the ALU0 **140** or the SHF0 **144** of the integer portion shown in FIG. **3**, which respectively perform non-multiplication integer operations and shift operations. Read multiplexers **112-1** and **112-3** also receive data from RFF1_IMED[ ] and RFF2_IMED[ ], respectively.

Each arithmetic-type unit **102–106** in the floating point portion receives two inputs, from respective sets of first and second source multiplexers S1 and S2. The first source of each unit ALU2, FALU, and MULT comes from the output of either read multiplexer **112-0** or **112-2**, and the second source comes from the output of either read multiplexer **112-1** or **112-3**. The sources of the FALU and the MULT may also come from the integer portion of FIG. **3** on bus **114**.

The results of the ALU2, FALU, and MULT are provided back to the write multiplexers **110** for storage into the floating point registers RF[ ], and also to the read multiplexers **112** for re-use as operands of subsequent operations. The FALU also outputs a signal FALU_BD indicating the Boolean result of a floating point comparison operation. FALU_BD is calculated directly from internal zero and sign flags of the FALU.

Null byte tester NULL **108** performs null byte testing operations upon an operand from a first source multiplexer, in one mode that of the ALU2. NULL **108** outputs a Boolean signal NULLB_BD indicating whether the thirty-two-bit first source operand includes a byte of value zero.

The outputs of read multiplexers **112-0, 112-1**, and **112-4** are provided to the integer portion (of FIG. **3**) on bus **118**. The output of read multiplexer **112-4** is also provided as STDT_FP[ ] store data to the floating point load/store unit **122**.

FIG. **5** illustrates further details of the control of the S1 and S2 multiplexers. As seen, in one embodiment, each S1 multiplexer may be responsive to bit B1 of the instruction I[ ], and each S2 multiplexer may be responsive to bit B2 of the instruction I[ ]. The S1 and S2 multiplexers select the sources for the various functional units. The sources may come from either of the register files, as controlled by the B1 and B2 bits of the instruction itself. Additionally, each register file includes two read ports from which the sources may come, as controlled by hardware not shown in the Figs.

B. Integer Portion Data Paths

As seen in FIG. **3**, the register set A **18** is also multi-ported. In one embodiment, the register set A **18** has two write ports WA0-1, and five read ports RDA0-4. The integer functional unit **66** of FIG. 1 is comprised of the ALU0 **140**, ALU1 **142**, SHF0 **144**, and NULL **146** of FIG. **3**. All elements of FIG. **3** except the register set **18** and the elements **140–146** comprise the SMC unit A of FIG. 1.

External data bus EX_DATA[ ] provides data to the integer load/store unit **152**. Immediate integer data on bus LDI_IMED[ ] are provided in response to a "load immediate" instruction. Other immediate integer data are provided on busses RFA1_IMED and RFA2_IMED in response to non-load immediate instructions, such as an "add immediate". Data are also provided on bus EX_SR_DT[ ] in response to a "special register move" instruction. Data may also arrive from the floating point portion (shown in FIG. **2**) on busses **116** and **118**.

The integer register set's two write ports WA0 and WA1 are coupled to write multiplexers **148-0** and **148-1**, respectively. The write multiplexers **148** receive data from: the FALU or MULT of the floating point portion (of FIG. **2**); the ALU0; the ALU1; the SHF0; either EX_SR DT[ ] or LDI_IMED[]; and EX_DATA[ ].

The integer register set's five read ports RDA0 to RDA4 are coupled to read multiplexers **150-0** to **150-4**, respectively. Each read multiplexer also receives data from: either EX-SR_DT[ ] or LDI_IMED[ ] on load immediate bypass bus **160**; a load external data bypass bus **154**, which allows external load data to skip the register set A; ALU0; ALU1; SHF0; and either the FALU or the MULT of the floating point portion (of FIG. **2**). Read multiplexers **150-1** and **150-3** also receive data from RFA1_IMED[ ] and RFA2_IMED[ ], respectively.

Each arithmetic-type unit **140–144** in the integer portion receives two inputs, from respective sets of first and second source multiplexers S1 and S2. The first source of ALU0 comes from either the output of read multiplexer **150-2**, or a thirty-two-bit wide constant zero (0000_{hex}) or floating point read multiplexer **112-4**. The second source of ALU0 comes from either read multiplexer **150-3** or floating point read multiplexer **112-1**. The first source of ALU1 comes from either read multiplexer **150-0** or IF_PC[ ]. IF_PC[ ] is used in calculating a return address needed by the instruction fetch unit (not shown), due to the IEU's ability to perform instructions in an out-of-order sequence. The second source of ALU1 comes from either read multiplexer **150-1** or CF_OFFSET[ ]. CF_OFFSET[ ] is used in calculating a return address for a CALL instruction, also due to the out-of-order capability.

The first source of the shifter SHF0 **144** is from either: floating point read multiplexer **112-0** or **112-4**; or any-integer read multiplexer **150**. The second source of SHF0 is from either: floating point read multiplexer **112-0** or **112-4**; or integer read multiplexer-**150-0**, **150-2**, or **150-4**. SHF0 takes a third input from a shift amount multiplexer (SA). The third input controls how far to shift, and is taken by the SA multiplexer from either: floating point read multiplexer **112-1**; integer read multiplexer **150-1** or **150-3**; or a five-bit wide constant thirty-one (11111_2 or 31_{10}). The shifter SHF0 requires a fourth input from the size multiplexer (S). The fourth input controls how much data to shift, and is taken by the S multiplexer from either: read multiplexer **150-1**; read multiplexer **150-3**; or a five-bit wide constant sixteen (10000_2 or 16_{10}).

The results of the ALU0, ALU1, and SHF0 are provided back to the write multiplexers **148** for storage into the integer registers RA[ ], and also to the read multiplexers **150** for re-use as operands of subsequent operations. The output of either ALU0 or SHF0 is provided on bus **120** to the floating point portion of FIG. **3**. The ALU0 and ALU1 also output signals ALU0_BD and ALU1_BD, respectively, indicating the Boolean results of integer comparison operations. ALU0_BD and ALU1_BD are calculated directly from the zero and sign flags of the respective functional units. ALU0 also outputs signals EX_TADR[ ] and EX_VM_ADR. EX_TADR[ ] is the target address generated for an absolute branch instruction, and is sent to the IFU (not shown) for fetching the target instruction. EX_VM_ADR[ ] is the virtual address used for all loads from memory and stores to memory, and is sent to the VMU (not shown) for address translation.

Null byte tester NULL **146** performs null byte testing operations upon an operand from a first source multiplexer.

In one embodiment, the operand is from the **ALU0**. NULL **146** outputs a Boolean signal NULLA_BD indicating whether the thirty-two-bit first source operand includes a byte of value zero.

The outputs of read multiplexers **150-0** and **150-1** are provided to the floating point portion (of FIG. **2**) on bus **114**. The output of read multiplexer **150-4** is also provided as STDT_INT[ ] store data to the integer load/store unit **152**.

A control bit PSR[7] is provided to the register set A **18**. It is this signal which, in FIG. **1**, is provided from the mode control unit **44** to the IEU mode integer switch **34** on line **46**. The IEU mode integer switch is internal to the register set A **18** as shown in FIG. **3**.

FIG. **6** illustrates further details of the control of the S1 and S2 multiplexers. The signal ALU0_BD.

C. Boolean Portion Data Paths

As seen in FIG. **4**, the register set C **22** is also multi-ported. In one embodiment, the register set C **22** has two write ports WC0-1, and five read ports RDA0-4. All elements of FIG. **4** except the register set **22** and the Boolean combinational unit **70** comprise the SMC unit C of FIG. **1**.

The Boolean register set's two write ports WC0 and WC1 are coupled to write multiplexers **170-0** and **170-1**, respectively. The write multiplexers **170** receive data from: the output of the Boolean combinational unit **70**, indicating the Boolean result of a Boolean combinational operation; ALU0_BD from the integer portion of FIG. **3**, indicating the Boolean result of an integer comparison; FALU_BD from the floating point portion of FIG. **2**, indicating the Boolean result of a floating point comparison; either ALU1_BD_P from ALU1, indicating the results of a compare instruction in ALU1, or NULLA_BD from NULL **146**, indicating a null byte in the integer portion; and either ALU2_BD_P from ALU2, indicating the results of a compare operation in ALU2, or NULLB_BD from NULL **108**, indicating a null byte in the floating point portion. In one mode, the ALU0_BD, ALU1_BD, ALU2_BD, and FALU_BD signals are not taken from the data paths, but are calculated as a function of the zero flag, minus flag, carry flag, and other condition flags in the PSR. In one mode, wherein up to eight instructions may be executing at one instant in the IEU, the IEU maintains up to eight PSRs.

The Boolean register set C is also coupled to bus EX_SR_DT[ ], for use with "special register move" instructions. The CSR may be written or read as a whole, as though it were a single thirty-two-bit register. This enables rapid saving and restoration of machine state information, such as may be necessary upon certain drastic system errors or upon certain forms of grand scale context switching.

The Boolean register set's five read ports RDC0 to RDC3 are coupled to read multiplexers **172-0** to **172-4**, respectively. The read multiplexers **172** receive the same set of inputs as the write multiplexers **170** receive. The Boolean combinational unit **70** receives inputs from read multiplexers **170-0** and **170-1**. Read multiplexers **172-2** and **172-3** respectively provide signals BLBP_CPORT and BLBP_DPORT. BLBP_CPORT is used as the basis for conditional branching instructions in the IEU. BLBP_DPORT is used in the "add with Boolean" instruction, which sets an integer register in the A or B set to zero or one (with leading zeroes), depending upon the content of a register in the C set. Read port RDC4 is presently unused, and is reserved for future enhancements of the Boolean functionality of the IEU.

IV. Conclusion

While the features and advantages of the present invention have been described with respect to particular embodiments thereof, and in varying degrees of detail, it will be

appreciated that the invention is not limited to the described embodiments. The following Claims define the invention to be afforded patent coverage.

We claim:

1. A processor, comprising:

an execution unit that performs at least one operation according to an instruction;

a first register set including a plurality of first registers each for holding integer data; and

a second register set including a plurality of second registers each for holding said integer data and for holding floating point data,

wherein said instruction specifies which of said first and second register sets is to be accessed, and wherein said execution unit accesses said first register set or said second register set as specified by said instruction, reads an operand value from either said first register second register set as specified by said instruction, and writes a result value to said first register set or said second register set as specified by said instruction.

2. The system of claim **1**, wherein said data of a first type is 32 bits.

3. The system of claim **1**, wherein said data of a second type is 64 bits.

4. The system of claim **1**, wherein said first and second register sets each have two write ports and five read ports.

5. The system of claim **1**, wherein said execution unit operates on integer data.

6. The system of claim **1**, wherein said instruction performs operations upon operands to generate results, each instruction specifying a respective source address for each operand and a destination address for said result of said instruction, each address specifying a register set and an offset.

7. The system of claim **1**, wherein said data stored in said second register set has a size that is different from the size of the register in which the data is contained.

8. The system of claim **7**, wherein said execution unit stores integer data within said second register set.

9. The system of claim **7**, wherein said execution unit selects the size of operand to read from, or write to, a floating-point register based upon the type of instruction being executed rather than upon the width of the register from which the integer operand is being read, or to which the integer result is being written.

10. The system of claim **1**, further comprising a floating point execution unit that operates on floating point data.

11. The system of claim **1** wherein said instruction indicates whether said first register set or said second register set should be used.

12. A method for efficiently utilizing register file resources, comprising the steps of:

(1) executing an instruction having two operands to produce an integer result, said operands are stored in a register file; wherein said register file includes an integer register set and a floating point register set; wherein said instruction indicates the location of said operands;

(2) accessing said integer register set or said floating point register set to retrieve said operands based on said instruction; and

(3) storing said integer result in said integer register set or said floating point register set based on said instruction.

* * * * *

EXHIBIT H

US005737624A

United States Patent [19]

Garg et al.

[11] **Patent Number: 5,737,624**

[45] **Date of Patent: *Apr. 7, 1998**

[54] **SUPERSCALAR RISC INSTRUCTION SCHEDULING**

[75] Inventors: **Sanjiv Garg**, Freemont; **Kevin Ray Iadonato**, San Jose; **Le Trong Nguyen**, Monte Sereno; **Johannes Wang**, Redwood City, all of Calif.

[73] Assignee: **Seiko Epson Corporation**, Tokyo, Japan

[*] Notice: The term of this patent shall not extend beyond the expiration date of Pat. No. 5,497,499.

[21] Appl. No.: **594,401**

[22] Filed: **Jan. 31, 1996**

Related U.S. Application Data

[63] Continuation of Ser. No. 219,425, Mar. 29, 1994, Pat. No. 5,497,499, which is a continuation of Ser. No. 860,719, Mar. 21, 1992, abandoned.

[51] **Int. Cl.**^{6} **G06F 9/38**

[52] **U.S. Cl.** **395/800.23**; 395/391; 395/393; 395/394

[58] **Field of Search** 395/800, 325, 395/391, 800.23, 392, 393, 381, 394

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,901,233	2/1990	Liptay	395/375
4,903,196	2/1990	Pomerene et al.	364/200
4,942,525	7/1990	Shintani et al.	395/375
4,992,938	2/1991	Cocke et al.	364/200
5,067,069	11/1991	Fite et al.	395/375
5,109,495	4/1992	Fite et al.	395/375
5,142,633	8/1992	Murray et al.	395/375
5,214,763	5/1993	Blaner et al.	395/375
5,222,244	6/1993	Carbine et al.	395/800
5,226,126	7/1993	McFarland et al.	395/375
5,251,306	10/1993	Tran	395/375
5,261,071	11/1993	Lyon	395/425
5,345,569	9/1994	Tran	395/375
5,355,457	10/1994	Shebanow et al.	395/375
5,398,330	3/1995	Johnson	395/375
5,448,705	9/1995	Nguyen et al.	395/375
5,487,156	1/1996	Popescu et al.	395/375
5,497,499	3/1996	Garg et al.	395/800

FOREIGN PATENT DOCUMENTS

0 515 166	11/1992	European Pat. Off. .
0 533 337	3/1993	European Pat. Off. .
WO91/20031	12/1991	WIPO .

OTHER PUBLICATIONS

"Critical Issues Regarding HPS, A High Performance Microarchitecture", Yale N. Patt, Stephen W. Melvin, Wen–Mei Hwu and Michael C. Shebanow; *The 18th Annual Workshop on Microprogramming*, Pacific Grove, California, Dec. 3–6, 1985, IEEE Computer Order No. 653, pp. 109–116.

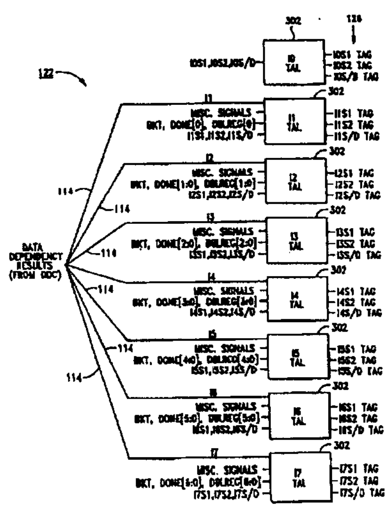
(List continued on next page.)

Primary Examiner—Larry D. Donaghue
Attorney, Agent, or Firm—Sterne, Kessler, Goldstein & Fox P.L.L.C.

[57] **ABSTRACT**

A register renaming system for out-of-order execution of a set of reduced instruction set computer instructions having addressable source and destination register fields, adapted for use in a computer having an instruction execution unit with a register file accessed by read address ports and for storing instruction operands. A data dependence check circuit is included for determining data dependencies between the instructions. A tag assignment circuit generates one of more tags to specify the location of operands, based on the data dependencies determined by the data dependence check circuit. A set of register file port multiplexers select the tags generated by the tag assignment circuit and pass the tags onto the read address ports of the register file for storing execution results.

19 Claims, 9 Drawing Sheets



OTHER PUBLICATIONS

"HPS, A New Microarchitecture: Rationale and Introduction", Yale N. Patt, Wen-Mei Hwu and Michael C. Shebanow; *The 18th Annual Workshop on Microprogramming*, Pacific Grove, California, Dec. 3-6, 1985; IEEE Computer Society Order No. 653, pp. 103-108.

Peleg et al., "Future Trends in Microprocessors: Out-Of-Order Execution, Spec. Branching and Their CISC Performance Potential", Mar. 1991.

Lightner et al., "The Metaflow Architecture", pp. 11-12, 63-68, IEEE Micro Magazine, Jun. 1991.

John L. Hennessy & David A Patterson, *Computer Architecture A Quantitative Approach*, Ch. 6.4, 6.7 and p. 449, 1990.

Hwu, Wen-mei, Steve Melvin, Mike Shebanow, Chein Chen, Jia-juin Wei, Yale Patt, "An HPS Implementation of VAX: Initial Design and Analysis", *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, pp. 282-291, 1986.

Hwu et al., "Experiments with HPS, a Restricted Data Flow Microarchitecture for High Performance Computers", *COMPCON 86*, 1986.

Hwu, Wen-mei and Yale N. Patt, "HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality", *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 297-306, Jun. 1986.

Yale N. Patt, Stephen W. Melvin, Wen-mei Hwu, Michael C. Shebanow, Chein Chen, Jiajuin Wei, "Run-Time Generation of HPS Microinstructions From a VAX Instruction Stream", *Proceedings of Micro 19 Workshop*, New York, New York, pp. 1-7, Oct., 1986.

Swenson, John A. and Yale N. Patt, "Hierarchical Registers for Scientific Computers", *St. Malo '88*, University of California at Berkeley, pp. 346-353, 1988.

Butler, Michael and Yale Patt, "An Improved Area-Efficient Register Alias Table for Implementing HPS", University of Michigan, Ann Arbor, Michigan, pp. 1-15, Jan., 1990.

Uvieghara, G.A., W.Hwu, Y. Nakagome, D.K. Jeong, D. Lee, D.A. Hodges, Y. Patt, "An Experimental Single-Chip Data Flow CPU", *Symposium on ULSI Circuits Design Digest of Technical Papers*, May, 1990.

Melvin, Stephen and Yale Patt, "Exploiting Fine-Grained Parallelism Through a Combination of Hardware and Software Techniques", *Proceedings From ISCA-18*, pp. 287-296, May, 1990.

Butler, Michael, Tse-Yu Yeh, Yale Patt, Mitch Alsup, Hunter Scales and Michael Shebanow, "Single Instruction Stream Parallelism Is Greater Than Two" *Proceedings of ISCA-18*, pp. 276-286, May, 1990.

Uvieghara, Gregory A., Wen-mei, W. Hwu, Yoshinobu Nakagome, Deog-Kyoon Jeong, David D. Lee, David A. Hodges and Yale Patt, "An Experimental Single-Chip Data Flow CPU", *IEEE Journal of Solid-State Circuits*, vol. 27, No. 1, pp. 17-28, Jan., 1992.

Gee, Jeff, Stephen W. Melvin, Yale N. Patt, "The Implementation Prolog via VAX 8600 Microcode", *Proceedings of Micro 19*, New York City, pp. 1-7, Oct., 1986.

Hwu, Wen-mei and Yale N. Patt, "Design Choices for the HPSm Microprocessor Chip", *Proceedings of the Twentieth Annual Hawaii International Conference on System Sciences*, pp. 330-336, 1987.

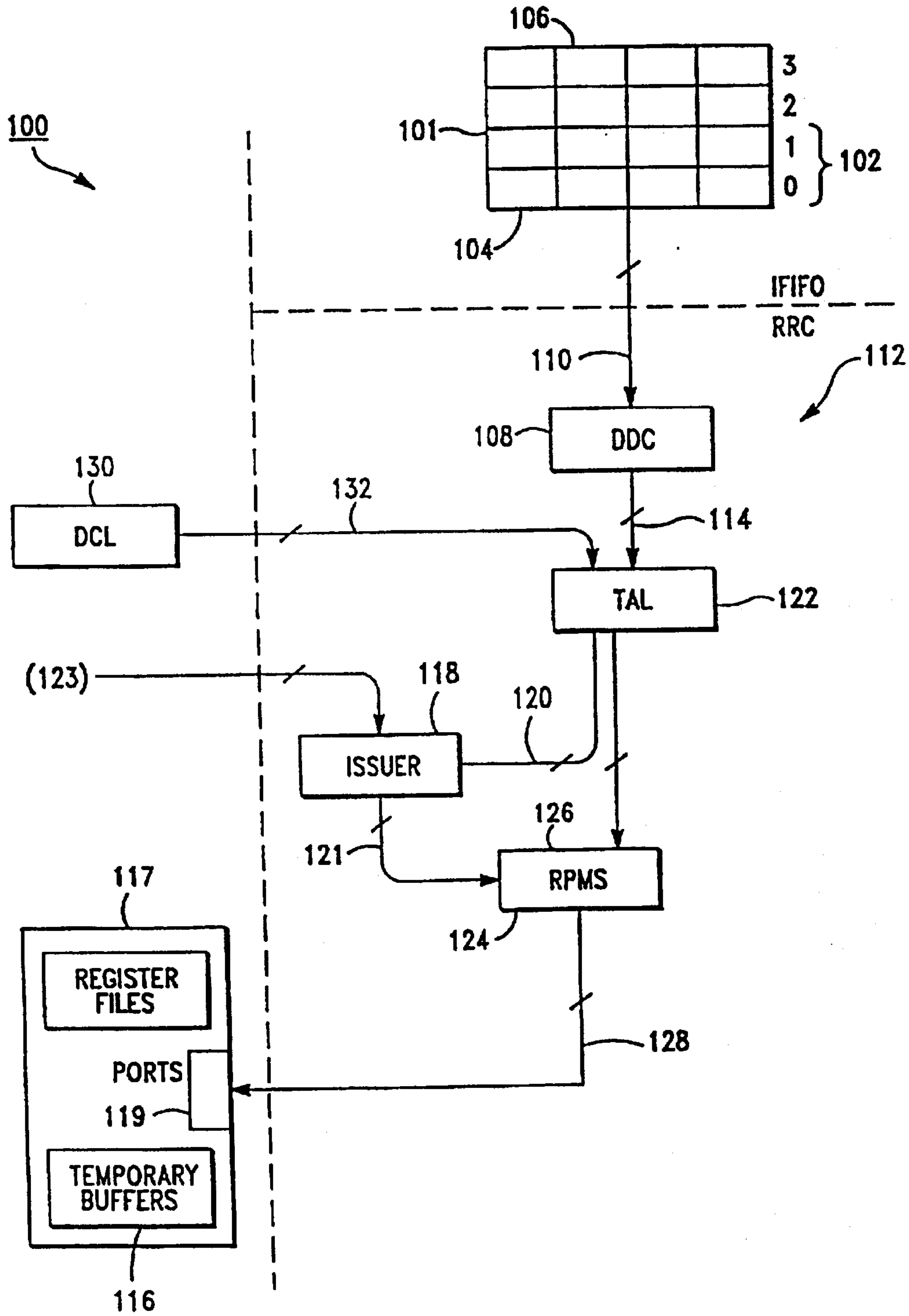
Wilson, James E., Steve Melvin, Michael Shebanow, Wen--mei Hwu and Yale N. Patt, "On Turning the Microarchitecture of an HPS Implementation of the VAX", *Proceedings of Micro 20*, pp. 162-167, Dec., 1987.

Hwu, Wen-mei and Yale N. Patt, "HPSm2: A Refined Single-chip Microengine", *HICSS '88*, pp. 30-40, 1988.

Keller, "Look-Ahead Processors"; Dec. 1975, pp. 177-194.

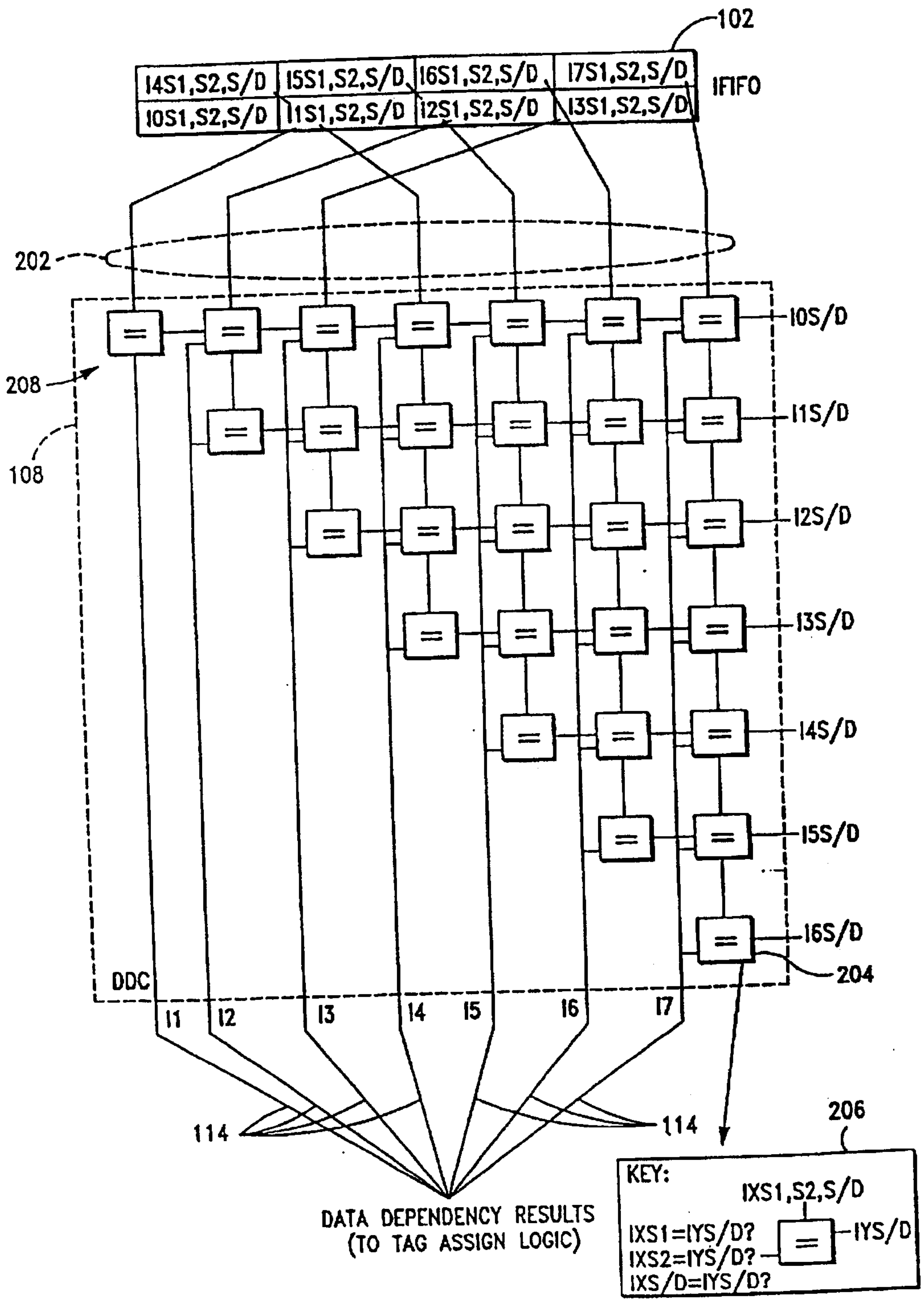
Dywer, A Multiple, Out-of-Order, Instruction Issuing System For SuperScaler Processors, (All); Aug. 1991.

Lightner et al., "The Metaflow Lightning" Chip Set Mar. 1991 *IEEE Lightning Outlined. Microprocessor Report* Sep. 1990.



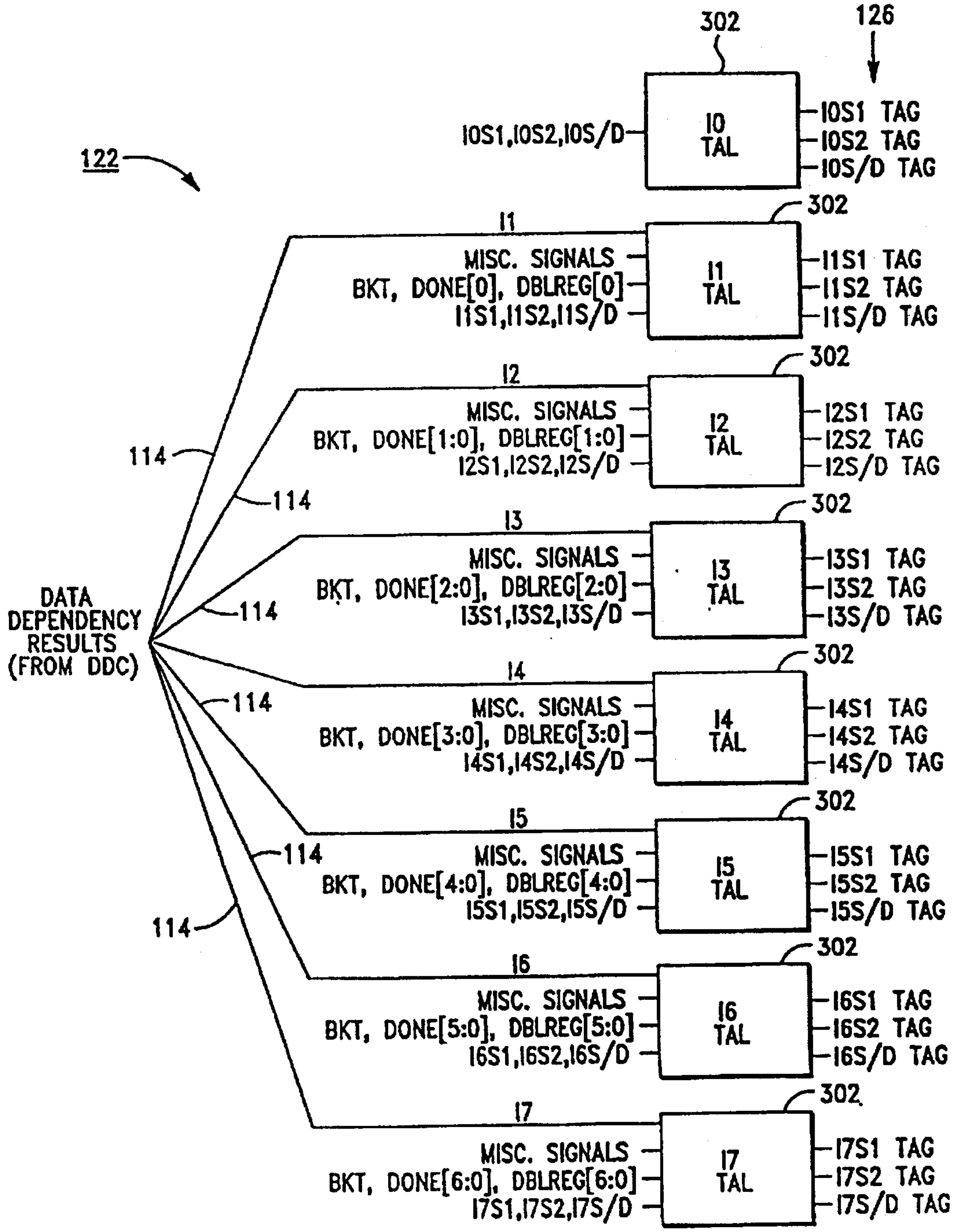
106
101
3
2
1
0
102
104
100
IFIFO
RRC
110
112
108
DDC
130
132
DCL
114
TAL
122
(123)
118
120
ISSUER
126
121
RPMS
117
124
REGISTER FILES
128
PORTS
119
TEMPORARY BUFFERS
116

FIG.—1



102
I4S1,S2,S/D
I5S1,S2,S/D
I6S1,S2,S/D
I7S1,S2,S/D
I0S1,S2,S/D
I1S1,S2,S/D
I2S1,S2,S/D
I3S1,S2,S/D
IFIFO
202
208
108
I0S/D
I1S/D
I2S/D
I3S/D
I4S/D
I5S/D
I6S/D
204
DDC
I1
I2
I3
I4
I5
I6
I7
114
114
206
DATA DEPENDENCY RESULTS
(TO TAG ASSIGN LOGIC)
KEY:
IXS1,S2,S/D
IXS1=IYS/D?
IXS2=IYS/D?
IXS/D=IYS/D?
IYS/D

FIG.—2



302
126
I0S1,I0S2,I0S/D
I0 TAL
I0S1 TAG
I0S2 TAG
I0S/D TAG
122
I1
MISC. SIGNALS
BKT, DONE[0], DBLREG[0]
I1S1,I1S2,I1S/D
I1 TAL
I1S1 TAG
I1S2 TAG
I1S/D TAG
I2
MISC. SIGNALS
BKT, DONE[1:0], DBLREG[1:0]
I2S1,I2S2,I2S/D
I2 TAL
I2S1 TAG
I2S2 TAG
I2S/D TAG
I3
MISC. SIGNALS
BKT, DONE[2:0], DBLREG[2:0]
I3S1,I3S2,I3S/D
I3 TAL
I3S1 TAG
I3S2 TAG
I3S/D TAG
DATA DEPENDENCY RESULTS (FROM DDC)
114
I4
MISC. SIGNALS
BKT, DONE[3:0], DBLREG[3:0]
I4S1,I4S2,I4S/D
I4 TAL
I4S1 TAG
I4S2 TAG
I4S/D TAG
I5
MISC. SIGNALS
BKT, DONE[4:0], DBLREG[4:0]
I5S1,I5S2,I5S/D
I5 TAL
I5S1 TAG
I5S2 TAG
I5S/D TAG
I6
MISC. SIGNALS
BKT, DONE[5:0], DBLREG[5:0]
I6S1,I6S2,I6S/D
I6 TAL
I6S1 TAG
I6S2 TAG
I6S/D TAG
I7
MISC. SIGNALS
BKT, DONE[6:0], DBLREG[6:0]
I7S1,I7S2,I7S/D
I7 TAL
I7S1 TAG
I7S2 TAG
I7S/D TAG

FIG.—3

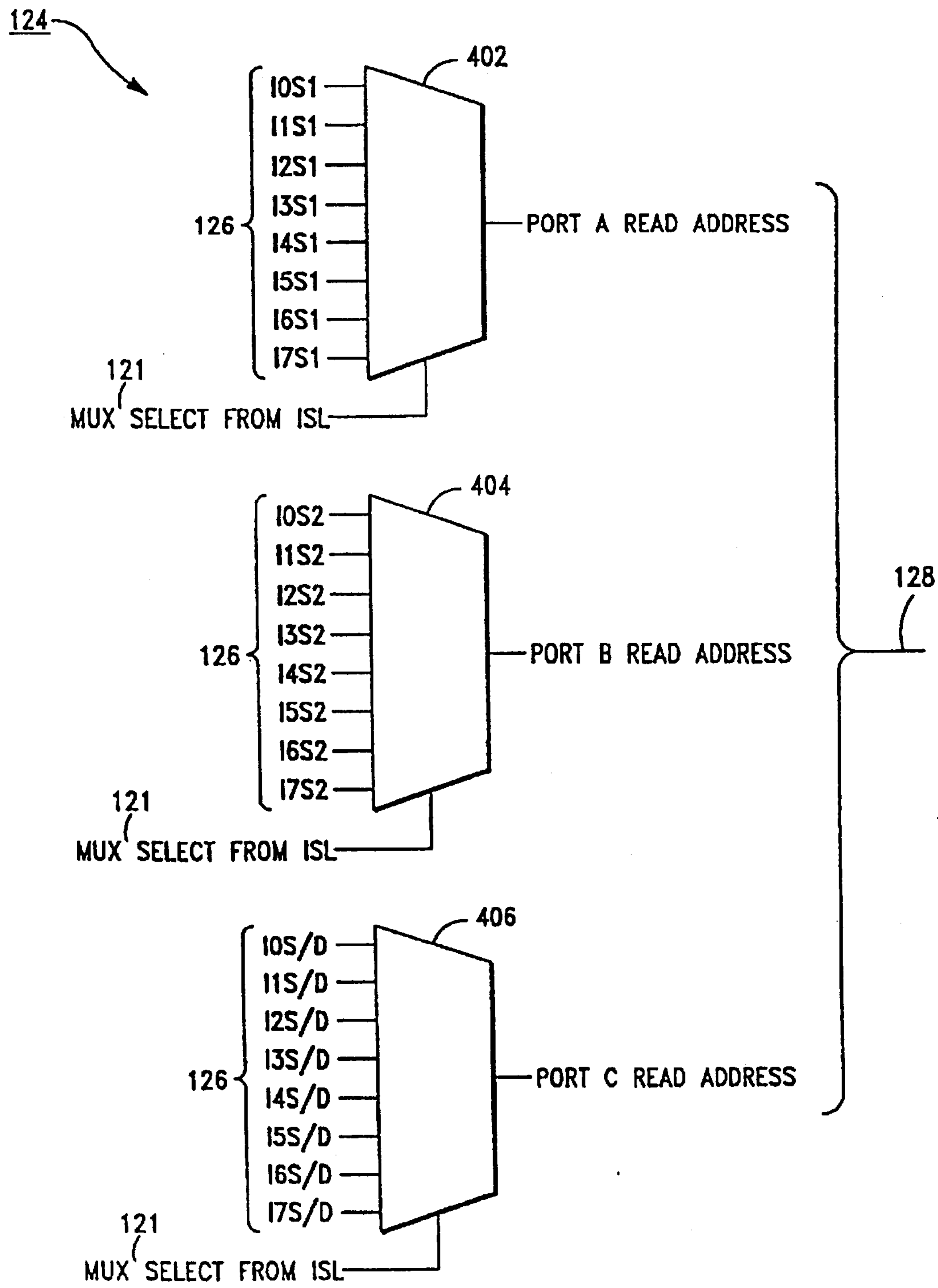

124
402
I0S1
I1S1
I2S1
I3S1
I4S1
I5S1
I6S1
I7S1
126
PORT A READ ADDRESS
121
MUX SELECT FROM ISL
404
I0S2
I1S2
I2S2
I3S2
I4S2
I5S2
I6S2
I7S2
126
PORT B READ ADDRESS
128
121
MUX SELECT FROM ISL
406
I0S/D
I1S/D
I2S/D
I3S/D
I4S/D
I5S/D
I6S/D
I7S/D
126
PORT C READ ADDRESS
121
MUX SELECT FROM ISL

FIG.—4

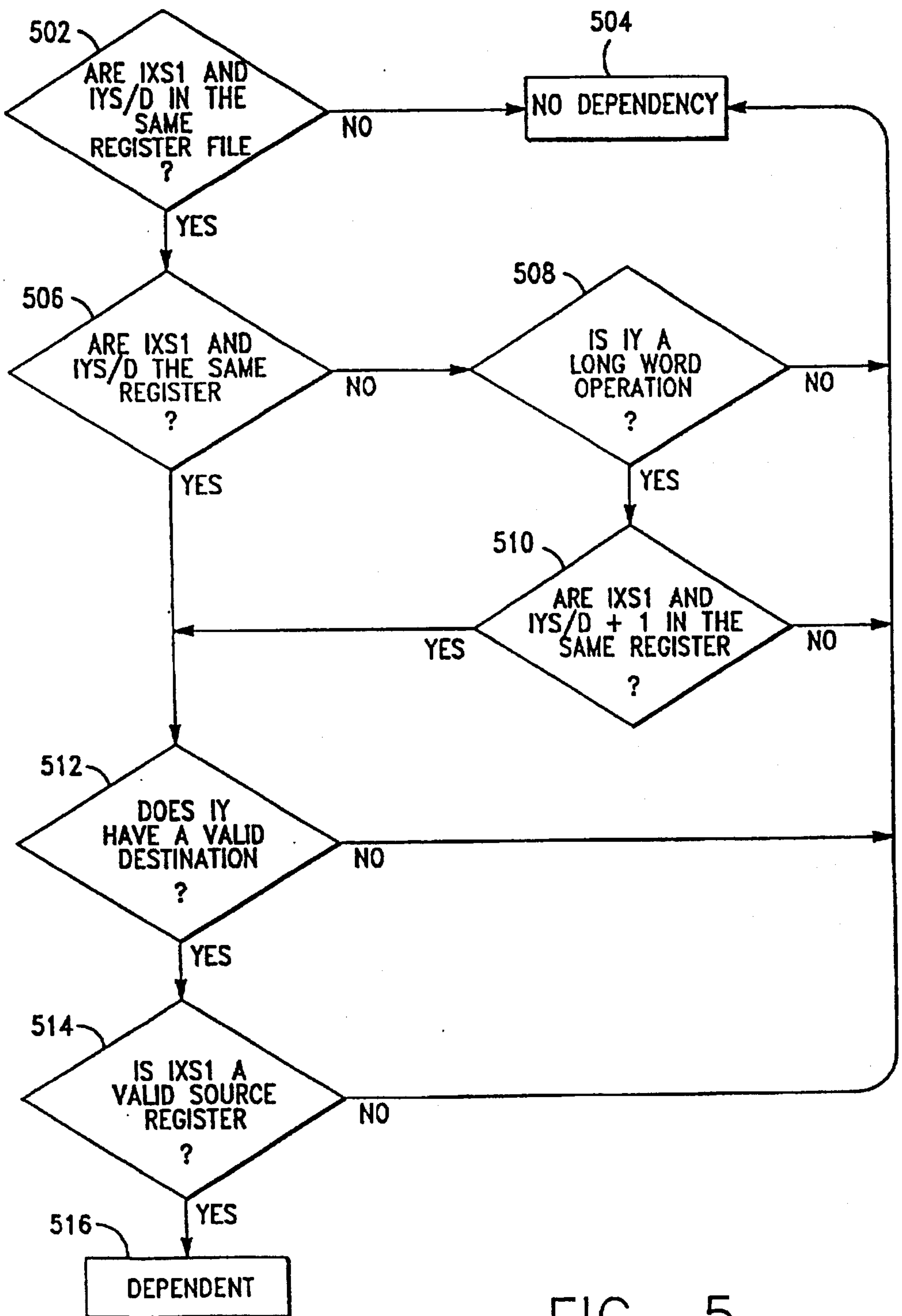

502
ARE IXS1 AND IYS/D IN THE SAME REGISTER FILE ?
NO
504
NO DEPENDENCY
YES
506
ARE IXS1 AND IYS/D THE SAME REGISTER ?
NO
508
IS IY A LONG WORD OPERATION ?
NO
YES
YES
510
ARE IXS1 AND IYS/D + 1 IN THE SAME REGISTER ?
YES
NO
512
DOES IY HAVE A VALID DESTINATION ?
NO
YES
514
IS IXS1 A VALID SOURCE REGISTER ?
NO
YES
516
DEPENDENT

FIG.—5

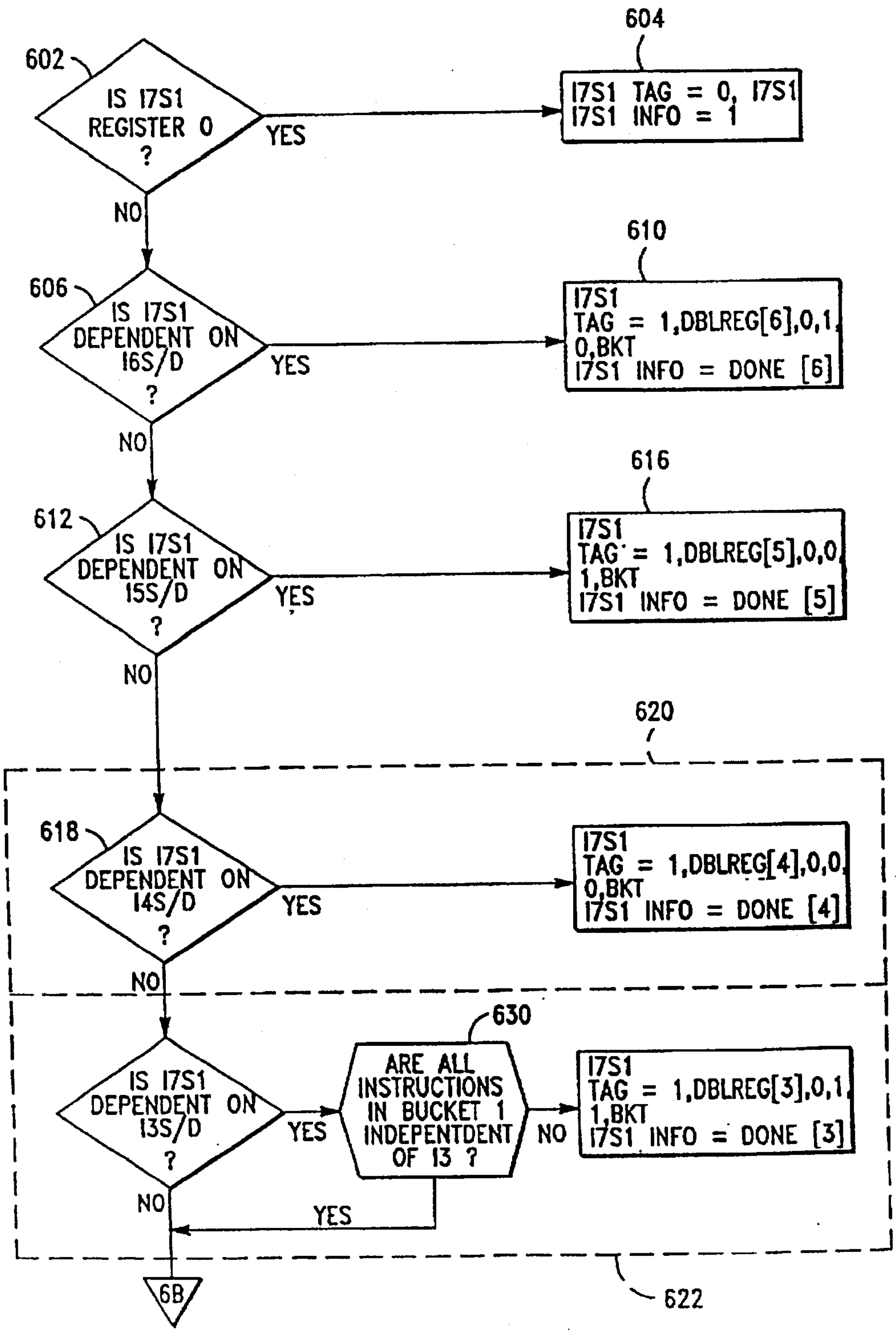

604
602
IS I7S1 REGISTER 0 ?
YES
I7S1 TAG = 0, I7S1
I7S1 INFO = 1
NO
610
606
IS I7S1 DEPENDENT ON I6S/D ?
YES
I7S1
TAG = 1,DBLREG[6],0,1,
0,BKT
I7S1 INFO = DONE [6]
NO
616
612
IS I7S1 DEPENDENT ON I5S/D ?
YES
I7S1
TAG = 1,DBLREG[5],0,0,
1,BKT
I7S1 INFO = DONE [5]
NO
620
618
IS I7S1 DEPENDENT ON I4S/D ?
YES
I7S1
TAG = 1,DBLREG[4],0,0,
0,BKT
I7S1 INFO = DONE [4]
NO
630
IS I7S1 DEPENDENT ON I3S/D ?
YES
ARE ALL INSTRUCTIONS IN BUCKET 1 INDEPENTDENT OF I3 ?
NO
I7S1
TAG = 1,DBLREG[3],0,1,
1,BKT
I7S1 INFO = DONE [3]
NO
YES
6B
622

FIG.—6A

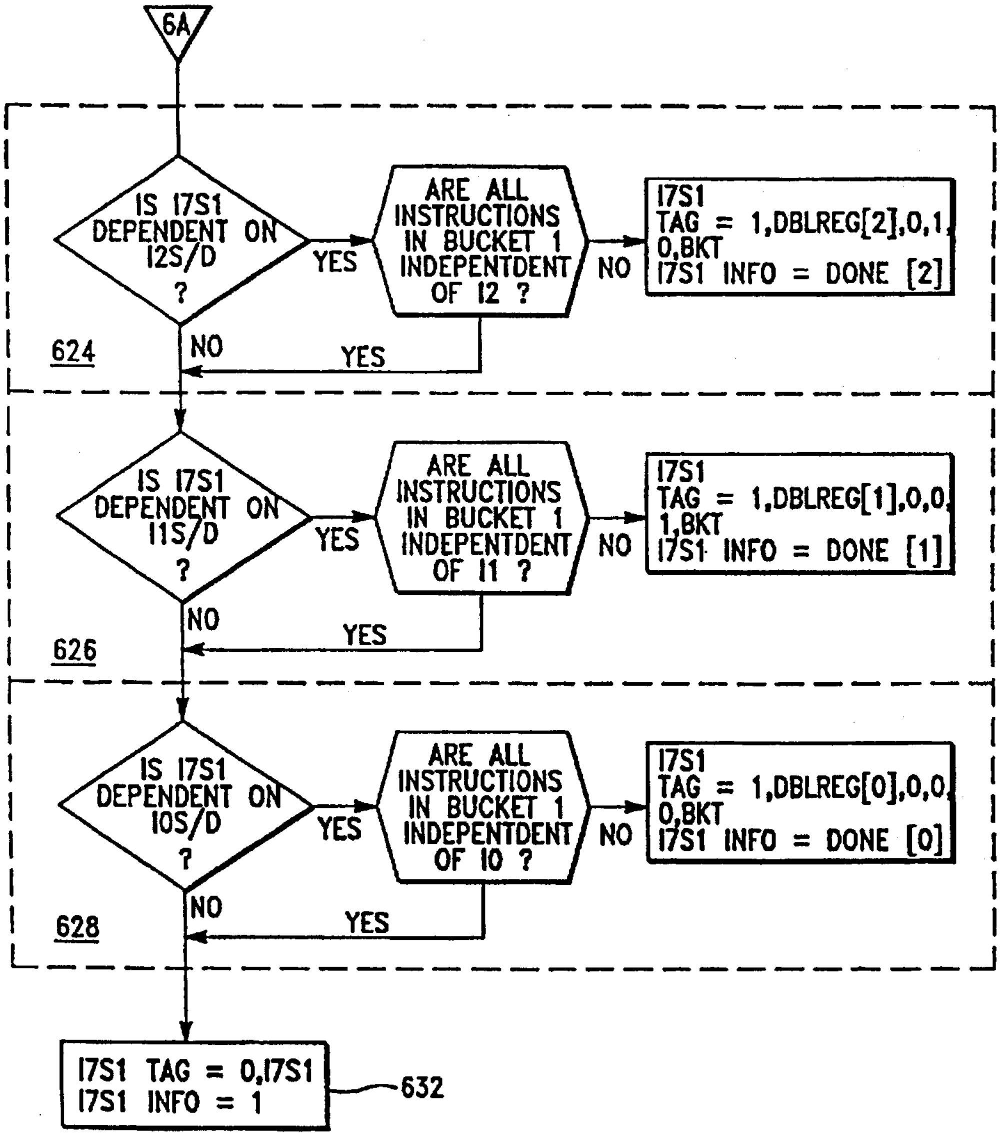

6A
IS I7S1 DEPENDENT ON I2S/D ?
YES
ARE ALL INSTRUCTIONS IN BUCKET 1 INDEPENTDENT OF I2 ?
NO
I7S1 TAG = 1,DBLREG[2],0,1, 0,BKT I7S1 INFO = DONE [2]
624
NO
YES
IS I7S1 DEPENDENT ON I1S/D ?
YES
ARE ALL INSTRUCTIONS IN BUCKET 1 INDEPENTDENT OF I1 ?
NO
I7S1 TAG = 1,DBLREG[1],0,0, 1,BKT I7S1 INFO = DONE [1]
626
NO
YES
IS I7S1 DEPENDENT ON I0S/D ?
YES
ARE ALL INSTRUCTIONS IN BUCKET 1 INDEPENTDENT OF I0 ?
NO
I7S1 TAG = 1,DBLREG[0],0,0, 0,BKT I7S1 INFO = DONE [0]
628
NO
YES
I7S1 TAG = 0,I7S1 I7S1 INFO = 1
632

FIG.—6B

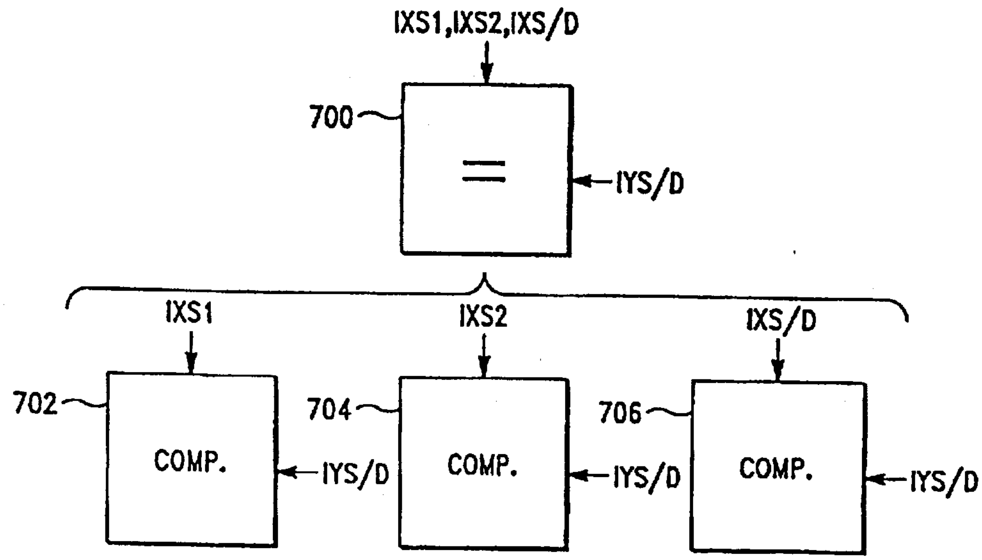

IXS1,IXS2,IXS/D
700
=
IYS/D
IXS1
IXS2
IXS/D
702
COMP.
IYS/D
704
COMP.
IYS/D
706
COMP.
IYS/D

FIG.—7

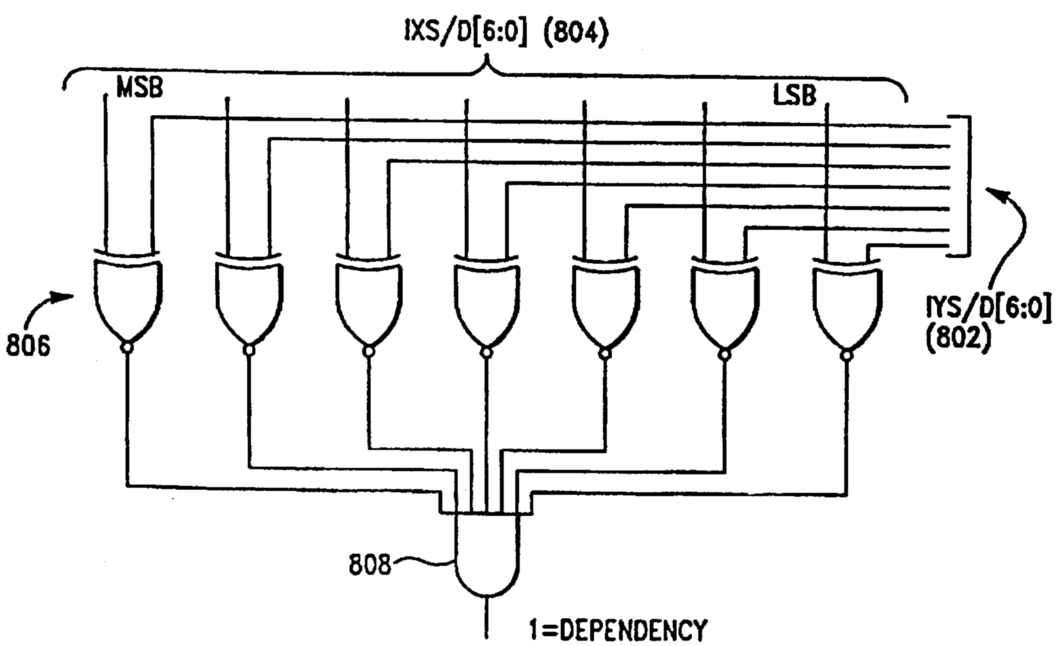

IXS/D[6:0] (804)
MSB
LSB
806
IYS/D[6:0]
(802)
808
1=DEPENDENCY

FIG.—8

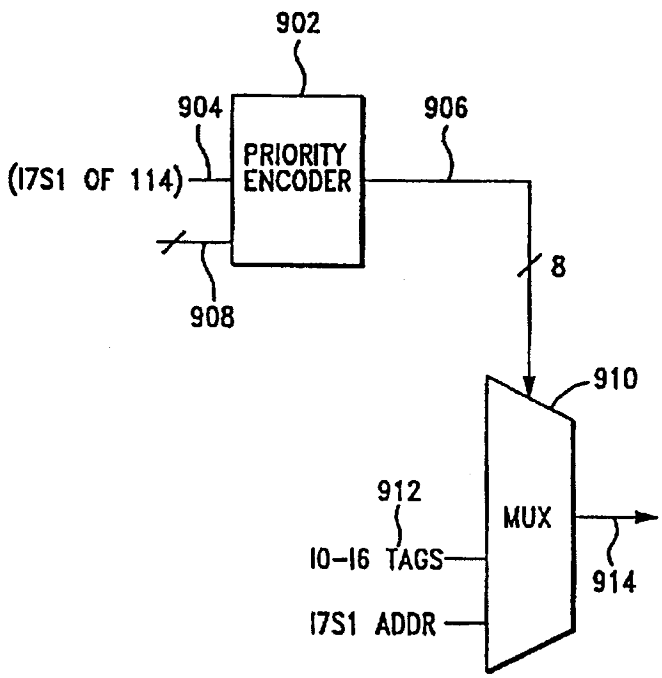

902
904
906
PRIORITY ENCODER
(I7S1 OF 114)
908
8
910
MUX
912
I0–I6 TAGS
I7S1 ADDR
914

FIG.—9

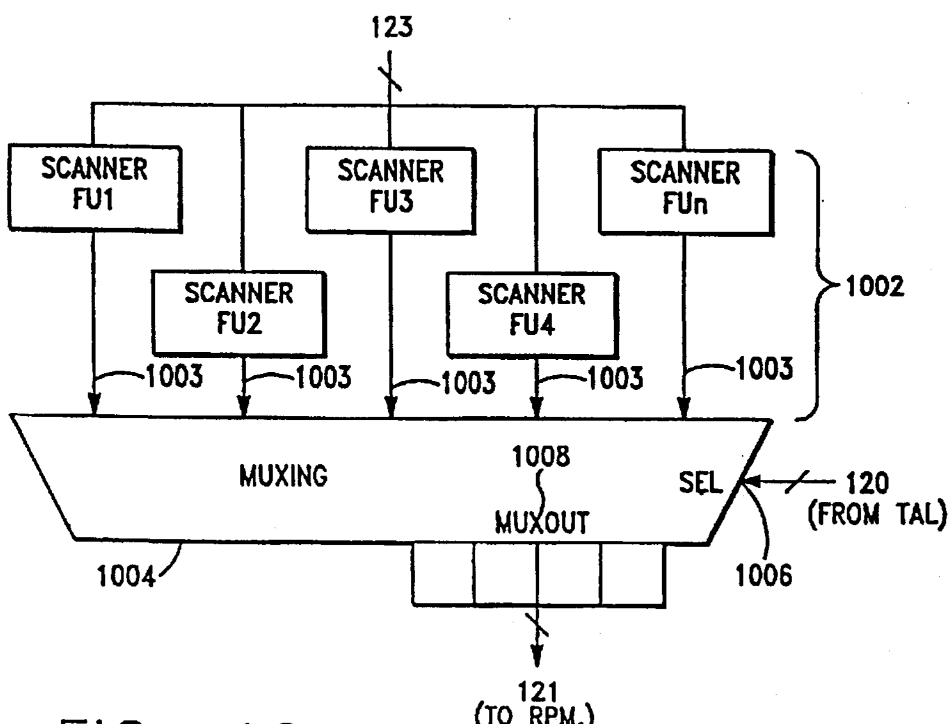

123
SCANNER FU1
SCANNER FU2
SCANNER FU3
SCANNER FU4
SCANNER FUn
1002
1003
1003
1003
1003
1003
MUXING
1008
MUXOUT
SEL
120
(FROM TAL)
1004
1006
121
(TO RPM.)

FIG.—10

SUPERSCALAR RISC INSTRUCTION SCHEDULING

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is a continuation of appl. Ser. No. 08/219,425, filed Mar. 29, 1994, now U.S. Pat. No. 5,497,499, which is a continuation of application Ser. No. 07/860,719, filed Mar. 21, 1992, now abandoned.

The following are commonly owned, applications:

"Semiconductor Floor Plan and Method for a Register Renaming Circuit", application Ser. No. 07/860,718, now U.S. Pat. No. 5,371,684, concurrently filed with the present application;

"High Performance RISC Microprocessor Architecture", Ser. No. 07/817,810, filed Jan. 8, 1992, now U.S. Pat. No. 5,539,911.

"Extensible RISC Microprocessor Architecture", Ser. No. 07/817,809, filed Jan. 8, 1992, abandoned in favor of FWC Application No. 08/397,016, (filed Mar. 1, 1995), now U.S. Pat. No. 5,560,032.

The disclosures of the above applications are incorporated herein by reference.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to superscalar reduced instruction set computers (RISC), more particularly, the present invention relates to instruction scheduling including register renaming and instruction issuing for superscalar RISC computers.

2. Related Art

A more detailed description of some of the basic concepts discussed in this application is found in a number of references, including Mike Johnson, Superscalar Microprocessor Design (Prentice-Hall, Inc., Englewood Cliffs, N.J., 1991); John L. Hennessy et al., "Computer Architecture—A Quantitative Approach" (Morgan Kaufmann Publishers, Inc., San Mateo, Calif., 1990). Johnson's text, particularly Chapters 2, 6 and 7 provide an excellent discussion of the register renaming issues addressed by the present invention.

A major consideration in a superscalar RISC processor is to how to execute multiple instructions in parallel and out-of-order, without incurring data errors due to dependencies inherent in such execution. Data dependency checking, register renaming and instruction scheduling are integral aspects of the solution.

2.1 Storage Conflicts and Register Renaming

True dependencies (sometimes called "flow dependencies" or "write-read" dependencies) are often grouped with anti-dependencies (also called "read-write" dependencies) and output dependencies (also called "write-write" dependencies) into a single group of instruction dependencies. The reason for this grouping is that each of these dependencies manifests itself through use of registers or other storage locations. However, it is important to distinguish true dependencies from the other two. True dependencies represent the flow of data and information through a program. Anti- and output dependencies arise because, at different points in time, registers or other storage locations hold different values for different computations.

When instructions are issued in order and complete in order, there is a one-to-one correspondence between registers and values. At any given point in execution, a register identifier precisely identifies the value contained in the corresponding register. When instructions are issued out of order and complete out of order, correspondence between registers and values breaks down, and values conflict for registers. This problem is severe when the goal of register allocation is to keep as many values in as few registers as possible. Keeping a large number of values in a small number of registers creates a large number of conflicts when the execution order is changed from the order assumed by the register allocator.

Anti- and output dependencies are more properly called "storage conflicts" because reusing storage locations (including registers) causes instructions to interfere with one another even though conflicting instructions are otherwise independent. Storage conflicts constrain instruction issue and reduce performance. But storage conflicts, like other resource conflicts, can be reduced or eliminated by duplicating the troublesome resource.

2.2 Dependency Mechanisms

Johnson also discusses in detail various dependency mechanisms, including: software, register renaming, register renaming with a reorder buffer, register renaming with a future buffer, interlocks, the copying of operands in the instruction window to avoid dependencies, and partial renaming.

A conventional hardware implementation relies on software to enforce dependencies between instructions. A compiler or other code generator can arrange the order of instructions so that the hardware cannot possibly see an instruction until it is free of true dependencies and storage conflicts. Unfortunately, this approach runs into several problems. Software does not always know the latency of processor operations, and thus, cannot always know how to arrange instructions to avoid dependencies. There is the question of how the software prevents the hardware from seeing an instruction until it is free of dependencies. In a scalar processor with low operation latencies, software can insert "no-ops" in the code to satisfy data dependencies without too much overhead. If the processor is attempting to fetch several instructions per cycle, or if some operations take several cycles to complete, the number of no-ops required to prevent the processor from seeing dependent instructions rapidly becomes excessive, causing an unacceptable increase in code size. The no-ops use a precious resource, the instruction cache, to encode dependencies between instructions.

When a processor permits out-of-order issue, it is not at all clear what mechanism software should use to enforce dependencies. Software has little control over the behavior of the processor, so it is hard to see how software prevents the processor from decoding dependent instructions. The second consideration is that no existing binary code for any scalar processor enforces the dependencies in a superscalar processor, because the mode of execution is very different in the superscalar processor. Relying on software to enforce dependencies requires that the code be regenerated for the superscalar processor. Finally, the dependencies in the code are directly determined by the latencies in the hardware, so that the best code for each version of a superscalar processor depends on the implementation of that version.

On the other hand, there is some motivation against hardware dependency techniques, because they are inherently complex. Assuming instructions with two input operands and one output value, as holds for typical RISC instructions, then there are five possible dependencies between any two instructions: two true dependencies, two

anti-dependencies, and one output dependency. Furthermore, the number of dependencies between a group of instructions, such as a group of instructions in a window, varies with the square of the number of instructions in the group, because each instruction must be considered against every other instruction.

Complexity is further multiplied by the number of instructions that the processor attempts to decode, issue, and complete in a single cycle. These actions introduce dependencies. The only aid in reducing complexity is that the dependencies can be determined incrementally, over many cycles to help reduce the scope and complexity of the dependency hardware.

One technique for removing storage conflicts is by providing additional registers that are used to reestablish the correspondence between registers and values. The additional registers are conventionally allocated dynamically by hardware, and the registers are associated with values needed by the program using "register renaming." To implement register renaming, processors typically allocate a new register for every new value produced (i.e., for every instruction that writes a register). An instruction identifying the original register, for the purpose of reading its value, obtains instead the value in the newly allocated register. Thus, hardware renames the original register identifier in the instruction to identify the new register and correct value. The same register identifier in several different instructions may access different hardware registers, depending on the locations of register references with respect to register assignments.

Consider the following code sequence where "op" is an operation, "Rn" represents a numbered register, and ":=" represents assignment:

$R3b:=R3a$ op $R5a$ (1)

$R4b:=R3b+1$ (2)

$R3c:=R5a+1$ (3)

$R7b:=R3c$ op $R4b$ (4)

Each assignment to a register creates a new "instance" of the register, denoted by an alphabetic subscript. The creation of a new instance for R3 in the third instruction avoids the anti- and output dependencies on the second and first instructions, respectively, and yet does not interfere with correctly supplying an operand to the fourth instruction. The assignment to R3 in the third instruction supersedes the assignment to R3 in the first instruction, causing R3c to become the new R3 seen by subsequent instructions until another instruction assigns a value to R3.

Hardware that performs renaming creates each new register instance and destroys the instance when its value is superseded and there are no outstanding references to the value. This removes anti- and output dependencies and allows more instruction parallelism. Registers are still reused, but reuse is in line with the requirements of parallel execution. This is particularly helpful with out-of-order issue, because storage conflicts introduce instruction issue constraints that are not really necessary to produce correct results. For example, in the preceding instruction sequence, renaming allows the third instruction to be issued immediately, whereas, without renaming, the instruction must be delayed until the first instruction is complete and the second instruction is issued.

Another technique for reducing dependencies is to associate a single bit (called a "scoreboard bit") with each register. The scoreboard bit is used to indicate that a register has a pending update. When an instruction is decoded that will write a register, the processor sets the associated scoreboard bit. The scoreboard bit is reset when the write actually occurs. Because there is only one scoreboard bit indicating whether or not there is a pending update, there can be only one such update for each register. The scoreboard stalls instruction decoding if a decoded instruction will update a register that already has a pending update (indicated by the scoreboard bit being set). This avoids output dependencies by allowing only one pending update to a register at any given time.

Register renaming, in contrast, uses multiple-bit tags to identify the various uncomputed values, some of which values may be destined for the same processor register (that is, the same program-visible register). Conventional renaming requires hardware to allocate tags from a pool of available tags that are not currently associated with any value and requires hardware to free the tags to the pool once the values have been computed. Furthermore, since scoreboarding allows only one pending update to a given register, the processor is not concerned about which update is the most recent.

A further technique for reducing dependencies is using register renaming with a "reorder buffer" which uses associative lookup. The associative lookup maps the register identifier to the reorder buffer entry as soon as the entry is allocated, and, to avoid output dependencies, the lookup is prioritized so that only the value for the most recent assignment is obtained if the register is assigned more than once. A tag is obtained if the result is not yet available. There can be as many instances of a given register as there are reorder buffer entries, so there are no storage conflicts between instructions. The values for the different instances are written from the reorder buffer to the register file in sequential order. When the value for the final instance is written to the register file, the reorder buffer no longer maps the register; the register file contains the only instance of the register, and this is the most recent instance.

However, renaming with a reorder buffer relies on the associative lookup in the reorder buffer to map register identifiers to values. In the reorder buffer, the associative lookup is prioritized so that the reorder buffer always provides the most recent value in the register of interest (or a tag). The reorder buffer also writes values to the register file in order, so that, if the value is not in the reorder buffer, the register file must contain the most recent value.

In a still further technique for reducing dependencies, associative lookup can be eliminated using a "future file." The future file does not have the properties of the reorder buffer discussed in the preceding paragraph. A value presented to the future file to be written may not be the most recent value destined for the corresponding register, and the value cannot be treated as the most recent value unless it actually is. The future file therefore keeps track of the most recent update and checks that each write corresponds to the most recent update before it actually performs the write.

When an instruction is decoded, it accesses tags in the future file along with the operand values. If the register has one or more pending updates, the tag identifies the update value required by the decoded instruction. Once an instruction is decoded, other instructions may overwrite this instructions's source operands without being constrained by anti-dependencies, because the operands are copied into the instruction window. Output dependencies are handled by preventing the writing as a result into the future file if the result does not have a tag for the most recent value. Both

anti- and output dependencies are handled without stalling instruction issue.

if dependencies are not removed through renaming, "interlocks" must use to enforce dependencies. An interlock simply delays the execution of an instruction until the instruction is free of dependencies. There are two ways to prevent an instruction from being executed: one way is to prevent the instruction from being decoded, and the other is to prevent the instruction from being issued.

To improve performance over scoreboarding, interlocks are moved from the decoder to the instruction window using a "dispatch stack." The dispatch stack is an instruction window that augments each instruction in the window with dependency counts. There is a dependency count associated with the source register of each instruction in the window, giving the number of pending prior updates to the source register and thus the number of updates that must be completed before all possible true dependencies are removed. There are two similar dependency counts associated with the destination register of each instruction in the window, giving both the number of pending prior uses of the register (which is the number of anti-dependencies) and the number of pending prior updates to the register (which is the number of output dependencies).

When an instruction is decoded and loaded into the dispatch stack, the dependency counts are set by comparing the instruction's register identifiers with the register identifiers of all instructions already in the dispatch stack. As instructions complete, the dependency counts of instructions that are still in the window are decremented based on the source and destination register identifiers of completing instructions (the counts are decremented by a variable amount, depending on the number of instructions completed). An instruction is independent when all of its counts are zero. The use of counts avoids having to compare all instructions in the dispatch stack to all other instructions on every cycle.

Anti-dependencies can be avoided altogether by copying operands to the instruction window (for example, to the reservation stations) during instruction decode. In this manner, the operands cannot be overwritten by subsequent register updates. Operands can be copied to eliminate anti-dependencies in any approach, independent of register renaming. The alternative to copying operands is to interlock anti-dependencies, but the comparators and/or counters required for these interlocks are costly, considering the number of combinations of source and result registers to be compared.

A tag can be supplied for the operand rather than the operand itself. This tag is simply a means for the hardware to identify which value the instruction requires, so that, when the operand value is produced, it can be matched to the instruction. If there can be only one pending update to a register, the register identifier can seive as a tag (as with scoreboarding). If there can be more than one pending update to a register (as with renaming), there must be a mechanism for allocating result tags and insuring uniqueness.

An alternative to scoreboarding interlocking is to allow multiple pending updates of registers to avoid stalling the decoder for output dependencies, but to handle anti-dependencies by copying operands (or tags) during decode. An instruction in the window is not issued until it is free of output dependencies, so the updates to each register are performed in the same order in which they would be performed with in-order completion, except that updates for different registers are out of order with respect to each other.

This alternative has almost all of the capabilities of register renaming, lacking only the capability to issue instructions so that updates to the same register occur out of order.

There appears to be no better alternative to renaming other than with a reorder buffer. Underlying the discussion of dependencies has been the assumption that the processor performs out-of-order issue and already has a reorder buffer for recovering from mispredicted branches. Out-of-order issue makes it unacceptable to stall the decoder for dependencies. If the processor has an instruction window, it is inconsistent to limit the look ahead capability of the processor by interlocking the decoder. There are then only two alternatives: implement anti- and output dependency interlocks in the window or remove these altogether with renaming.

SUMMARY OF THE INVENTION

The present invention is directed to instruction scheduling including register renaming and instruction issuing for superscalar RISC computers. A Register Rename Circuit (RRC), which is part of the scheduling logic allows a computer's Instruction Execution Unit (IEU) to execute several instructions at the same time while avoiding dependencies. In contrast to conventional register renaming, the present invention does not actually rename register addresses. The RRC of the present invention temporarily buffers the instruction results, and the results of out-of-order instruction execution are not transferred to the register file until all previous instructions are done. The RRC also performs result forwarding to provide temporarily buffered operands (results) to dependant instructions. The RRC contains three subsections: a Data Dependency Checker (DDC), Tag Assign Logic (TAL) and Register file Port MUXes (RPM).

The function of the DDC is to locate the dependencies between the instructions for a group of instructions. The DDC does this by comparing the addresses of the source registers of each instruction to the addresses of the destination registers of each previous instruction in the group. For example, if instruction A reads a value from a register that is written to by instruction B, then instruction A is dependent upon instruction B and instruction A cannot start until instruction B has finished. The DDC outputs indicate these dependencies.

The outputs of the DDC go to the TAL. Because it is possible for an instruction to be dependent on more than one previous instruction, the TAL must determine which of those previous instructions will be the last one to be executed. The present invention automatically maps each instruction a predetermined temporary buffer location; hence, the present invention does not need prioritized associative look-up as used by convention reorder buffers, thereby saving chip area/cost and execution speed.

Out-of-order results for several instructions being executed at the same time are stored in a set of temporary buffers, rather that the file register designated by the instruction. If the DDC determines, for example, that a register that instruction 6's source is written to by instructions 2, 3 and 5, then the TAL will indicate that instruction 6 must wait for instruction 5 by outputting the "tag" of instruction 5 for instruction 6. The tag of instruction 5 shows the temporary buffer location where instruction 5's result is stored. It also contains a one bit signal (called a "done flag") that indicates if instruction 5 is finished or not. The TAL will output three tags for each instruction, because each instruction can have three source registers. If an instruction is not dependent on

any previous instruction, the TAL will output the register file address of the instruction's input, rather an a temporary buffer's address.

The last part of the RRC are the RPMs or Register file Port MUXes. The inputs of the RPMs are the outputs of the TAL, and the select lines for the RPMs come from another part of the IEU called the Instruction Scheduler or Issuer. The Instruction Scheduler chooses which instruction to execute (this decision is based partly on the done flags) and then uses the RPMs to select the tags of that instruction. These tags go to the read address ports of the computer's register files. In the previous example, once instruction 5 has finished, the Instruction Scheduler will start instruction 6. It will select the RPM so that the address of instruction 5's result (its tag) is sent to the register file, and the register file will make the result of instruction 5 available to instruction 6.

The foregoing and other features and advantages of the present invention will be apparent from the following more particular description of the preferred embodiments of the invention, as illustrated in the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention will be better understood if reference is made to the accompanying drawings.

FIG. 1 shows a representative high level block diagram of the register renaming circuit of the present invention.

FIG. 2 shows a representative block diagram of the data dependency check circuit of the present invention.

FIG. 3 shows a representative block diagram of the tag assignment logic of the present invention.

FIG. 4 shows a representative block diagram of the register port file multiplexers of the present invention.

FIG. 5 is a representative flowchart showing a data dependency check method for IXS1 and IYS/D in accordance with the present invention.

FIGS. 6A and 6B are representative flowcharts showing a tag assignment method in accordance with the present invention.

FIG. 7 shows a representative block diagram which compares an instruction Y's source/destination operand with each operand of an instruction X in accordance with an embodiment of the present invention.

FIG. 8 shows a representative circuit diagram for comparator block 706 of FIG. 7.

FIG. 9 shows a representative block diagram of a Priority Encoder in accordance with an embodiment of the present invention.

FIG. 10 shows a representative block diagram of the instruction scheduling logic of the present invention.

DETAILED DESCRIPTION

FIG. 1 shows a representative high level block diagram of an Instruction Execution Unit (IEU) 100 associated with the present invention. The goal of IEU 100 is to execute as many instructions as possible in the shortest amount of time. There are two basic ways to accomplish this: optimize IEU 100 so that each instruction takes as little time as possible or optimize IEU 100 so that it can execute several instructions at the same time.

Instructions are sent to IEU 100 from an Instruction Fetch Unit (IFU, not shown) through an instruction FIFO (first-in-first-out register stack storage device) 101 in groups of four called "buckets." IEU 100 can decode and schedule up to two buckets of instructions at one time. FIFO 101 stores

16 total instructions in four buckets labeled 0–3. IEU 100 looks at the an instruction window 102. In one embodiment of the present invention, window 102 comprises eight instructions (buckets 0 and 1). Every cycle IEU 100 tries to issue a maximum number of instructions from window 102. Window 102 functions as a instruction buffer register. Once the instructions in a bucket are executed and their results stored in the processor's register file (see block 117), the bucket is flushed out a bottom 104 and a new bucket is dropped in at a top 106.

In order to execute instructions in parallel or out of order, care must be taken so that the data that each instruction needs is available when the instruction needs it and also so that the result of each instruction is available for any future instructions that might need it. A Register Rename Circuit (RRC), which is part of the scheduling logic of the computer's IEU performs this function by locating dependencies between current instructions and then renaming the sources (inputs) of the instruction.

As noted above, there are three types of dependencies: input dependencies, output dependencies and anti-dependencies. Input dependencies occur when an instruction, call it A, that performs an operation on the result of a previous instruction, call it B. Output dependencies occur when the outputs of A and B are to be stored in the same place. Anti-dependencies occur when instruction A comes before B in the instruction stream and B's result will be stored in the same place as one of A's inputs.

Input dependencies are handled by not executing instructions until their inputs are available. RRC 112 is used to locate the input dependencies between current instructions and then to signal an Instruction Scheduler or Issuer 118 when all inputs for a particular instruction are ready. In order to locate these dependencies, RRC 112 compares the register file addresses of each instruction's inputs with the addresses of each previous instruction's output using a data dependency circuit (DDC) 108. If one instruction's input comes from a register where a previous instruction's output will be stored, then the latter instruction must wait for the former to finish.

This implementation of RRC 112 can check eight instructions at the same time, so a current instruction is defined as any one of those eight from window 102. It should become evident to those skilled in the art that the present invention can easily be adapted to check more or less instructions.

In one embodiment of the present invention, instructions can have from 0 to 3 inputs and 0 or 1 outputs. Most instructions' inputs and outputs come from, or are stored in, one of several register files. Each register file 117 (e.g., separate integer, floating and boolean register files) has 32 real entries plus the group of 8 temporary buffers 116. When an instruction completes, (The term "complete" means that the operation is complete and the operand is ready to be written to its destination register.) its result is stored in its preassigned location in the temporary buffers 116. Its result is later moved to the appropriate place in register file 117 after all previous instructions' results have been moved to their places in the register file. This movement of results from temporary buffers 116 to register file 117 is called "retirement" and is controlled by termination logic, as should become evident to those skilled in the art. More than one instruction may be retired at a time. Retirement comprises updating the "official state" of the machine including the computer's Program Counter, as will become evident to those skilled in the art. For example, if instruction I0 happens to complete directly before instruction I1, both

results can be stored directly into register file 117. But if instruction I3 then completes, its result must be stored in temporary buffer 116 until instruction I2 completes. By having IEU 100 store each instruction's result in its preassigned place in the temporary buffers 116, IEU 100 can execute instructions out of program order and still avoid the problems caused by output and anti-dependencies.

RRC 112 sends a bit map to an Instruction Scheduler 118 via a bus 120 indicating which instructions in window 102 are ready for issuing. Instruction decode logic (not shown) indicates to Issuer 118 the resource requirements for each instruction over a bus 123. For each resource in IEU 100 (e.g., each functional unit being an adder, multiplier, shifter, or the like), Issuer 118 scans this information and selects the first and subsequent instructions for issuing by sending issue signals over bus 121. The issue signals select a group of Register File Port MUXes (RPMs) 124 inside RRC 112 whose inputs are the addresses of each instruction's inputs.

Because the results may stay in temporary buffer 116 several cycles before going to register file 117, a mechanism is provided to get results from temporary buffer 116 before they go to register file 117, so the information can be used as operands for other instructions. This mechanism is called "result forwarding," and without it, Issuer 118 would not be able to issue instructions out of order. This result forwarding is done in register file 117 and is controlled by RRC 112. The control signals necessary for performing the result forwarding will be come evident to those skilled in the art, as should the random logic used for generating such control signals.

If an instruction is not dependent on any of the current instructions result forwarding is not necessary since the instruction's inputs are already in register file 117. When Issuer 118 decides to execute that instruction, RRC 112 tells register file 117 to output its data.

RRC 112 contains three subsections: a Data Dependency Checker (DDC) 108, Tag Assign Logic (TAL) 122 and Register File Port MUXes (RPM) 124. DDC 108 determines where the input dependencies are between the current instructions. TAL 122 monitors the dependencies for Issuer 118 and controls result forwarding. RPM 124 is controlled by Issuer 118 and directs the outputs of TAL 122 to the appropriate register file address ports 119. Instructions are passed to DDC 108 via bus 110. All source registers are compared with all previous destination registers for each instruction in window 102.

Each instruction has only one destination, which may be a double register in one embodiment. An instruction can only depend on a previous instruction and may have up to three source registers. There are various register file source and destination addresses that need to be checked against each other for any dependencies. As noted above, the eight bottom instructions corresponding to the lower two buckets are checked by DDC 108. All source register addresses are compared with all previous destination register addresses for the instructions in window 102.

For example, let's say a program has the following instruction sequence:

add R0, R1, R2	(0)
add R0, R2, R3	(1)
add R4, R5, R2	(2)
add R2, R3, R4	(3)

The first two registers in each instruction 0–3 are the source registers, and the last listed register in each instruction is the destination register. For example, R0 and R1 are the source registers for instruction 0 and R2 is the destination register. Instruction 0 adds the contents of registers 0 and 1 and stores the result in R2. For instructions 1–3 in this example, the following are the comparisons needed to evaluate all of the dependencies:

I1S1, I1S2 vs. I0D

I2S1, I2S2 vs. I1D, I0D

I3S1, I3S2 vs. I2D, I1D, I0D

The key to the above is as follows: IXRS1 is the address of source (input) number 1 of instruction X; IXRS2 is the address of source (input) number 2 of instruction X; and IXD is the address of the destination (output) of instruction X.

Note also that RRC 112 can ignore the fact that instruction 2 is output dependent on instruction 0, because the processor has a temporary buffer where instruction 2's result can be stored without interfering with instruction 0's result. As discussed before, instruction 2's result will not be moved from temporary buffers 116 to register file 117 until instructions 0 and 1's results are moved to register file 117.

The number of instructions that can be checked by RRC 112 is easily scaleable. In order to check eight instructions at a time instead of four, the following additional comparisons would also need to be made:

I4S1, I4S2 vs I3D, I2D, I1D, I0D

I5S1, I5S2 vs I4D, I3D, I2D, I1D, I0D

I6S1, I6S2 vs I5D, I4D, I3D, I2D, I1D, I0D

I7S1, I7S2 vs I6D, I5D, I4D, I3D, I2D, I1D, I0D

There are several special cases that RRC 112 must handle in order to do the dependency check. First, there are some instructions that use the same register as an input and an output. Thus, RRC 112 must compare this source/destination register address with the destination register addresses of all previous instructions. So for instruction 7, the following comparisons would be necessary:

I7S1,I7S2,I7S/D vs. I6D,I5D,I4D,I3D,I2D,I1D,I0D.

Another special case occurs when a program contains instructions that generate 64 bit outputs (called long-word operations). These instructions need two registers in which to store their results. In this embodiment, these registers must be sequential. Thus if RRC 112 is checking instruction 4's dependencies and instruction 1 is a long-word operation, then it must do the following comparisons:

I4S1,I4S2 vs. I3D,I2D,I1D,I1D+1,I0D

Sometimes, instructions do not have destination registers. Thus RRC 112 must ignore any dependencies between instructions without destination registers and any future instructions. Also, instructions may not have only one valid source register, so RRC 112 must ignore any dependencies between the unused source register (usually S2) and any previous instructions.

RRC 112 is also capable of dealing with multiple register files. When using multiple register files, dependencies only

occur when one instruction's source register has the same address and is in the same register file as some other instruction's destination register. RRC 112 treats the information regarding which register file a particular address is from as part of the address. For example, in an implementation using four 32 bit register files, RRC 112 would do 7 bit compares instead of 5 bit compares (5 for the address and 2 for the register file).

Signals indicating which instructions are long-word operations or have invalid source or destination registers are sent to RRC 112 from Instruction Decode Logic (IDL; not shown). IDL also tells RRC 112 which register file each instruction's sources and destinations will come from or go to.

A block diagram of DDC 108 is shown in FIG. 2. Source address signals arrive from IFIFO 101 for all eight instructions of window, 102. Additional inputs include long-word load operation flags, register file decode signals, invalid destination register flags, destination address signals and addressing mode flags for all eight instructions.

DDC 208 comprises 28 data dependency blocks 204. Each block 204 is described in a KEY 206. Each block 204 receives 3 inputs, IXS1, IXS2 and IXS/D. IXS1 is the address of source (input) number 1 of instruction X, IXS2 is the address of source (input) number 2 of instruction X; and IXS/D is the address of the source/destination (input) of instruction X. Each block 204 also receives input INS/D, which is the destination register address for some previous instruction Y. A top row 208, for example, receives I0S/D, which is the destination register address for instruction 0. Each block 204 outputs the data dependency results to one of a corresponding bus line 114. For example, the address of I2S/D must be checked with operand addresses S1, S2 and S/D of instructions 7, 6, 5, 4, and 3.

Each block 204 performs the three comparisons. To illustrate these comparisons, consider a generic block 700 shown in FIG. 7, which compares instruction Y's source/destination operand with each operand of instruction X. In this example, the three following comparisons must be made:

IXS1=IYS/D

IXS2=IYS/D

IXS/D=IYS/D

These comparisons are represented by three comparator blocks 702, 704 and 706, respectively. One set of inputs to comparator blocks 702, 704 and 706 are the bits of the IYS/D field, which is represented by number 708. Comparator block 702 has as its second set of inputs the bits of the IXS1. Similarly, comparator block 704 has as its second set of inputs the bits of the IXS1, and comparator block 706 has as its second set of inputs the bits of the IXS/D.

In a preferred embodiment, the comparisons performed by blocks 702, 704 and 706 can be performed by random logic. An example of random logic for comparator block 706 is shown in FIG. 8. Instruction Y's source/destination bits [6:0] are shown input from the right at reference number 802 and instruction X's source/destination bits [6:0] are shown input from the top at reference number 804. The most significant bit (MSB) is bit 6 and the least significant bit (LSB) is bit 0. The corresponding bits from the two operands are fed to a set of seven exclusive NOR gates (XNORs) 806. The outputs of XNORs 806 are then ANDed by a seven input AND gate 808. If the corresponding bits are the same,

the output of XNOR 806 will be logic high. When all bits are the same, all seven XNOR 806 outputs are logic high and the output of AND gate 808 is logic high, this indicates that there is a dependency between IXS/D and IYS/D.

The random logic for comparator blocks 702 and 704 will be identical to that shown in FIG. 8. The present invention contemplates many other random logic circuits for performing data dependency checking, as will become evident to those skilled in the art without departing from the spirit of this example.

As will further become evident to those skilled in the art, various implementation specific special cases can arise which require additional random logic to perform data dependency checking. An illustrative special data dependency checking case is for long word handling.

As mentioned before, if a long word operation writes to register X, the first 32 bits are written to register X and the second 32 bits are written to register X+1. The data dependency checker therefore needs to check both resisters when doing a comparison. In a preferred embodiment, register X is an even register, X+1 is an odd register and thus they only differ by the LSB. The easiest way to check both registers at the same time is to simply ignore the LSB. In the case of a store long (STLG) or load long (LDLG) operation, if X and Y only differ by the LSB bit [0], the logic in FIG. 8 would cause there to be no dependency, when there really is a dependency. Therefore, for a long word operation the STLG and LDLG flags must be ORed with the output of the [0] bit XNOR to assure that all dependencies are detected.

A data dependency check flowchart for IXS1 and IYS/D is shown in FIG. 5. DDC 108 first checks whether IXS1 and IYS/D are in the same register file, as shown at a conditional block 502. If they are not in the same register file there is no dependency. This is shown at block a 504. if there is a dependency, DDC 108 then determines whether IXS1 and IYS/D are in the same register, as shown at a block 506. If they are not in the same register, flow proceeds to a conditional block 508 where DDC 108 determines whether IY is a long word operation. If IY is not a long word operation there is no dependency and flow proceeds to a block 504. If IY is a long word operation, flow then proceeds to a conditional statement 510 where DDC 108 determines whether IXS1 and IYS/D+1 are the same register. If they are not, there is no dependency and flow proceeds to a block 504. If IXS1 and IYS/D+1 are the same register, flow proceeds to a conditional block 512 where DDC 108 determines if IY has a valid destination. If it does not have a valid destination, there is no dependency and flow proceeds to block 504. If IY does have a valid destination, flow proceeds to a conditional block 514 where DDC 108 determines if IXS1 has a valid source register. Again, if no valid source register is detected there is no dependency, and flow proceeds to a block 504. If a valid source register is detected, DDC 108 has determined that there is a dependency between IXS1 and IYX/D, as shown at a block 516.

A more derailed discussion of dam dependency checking is found in commonly owned, copending application application Ser. No. 07/860,718, now U.S. Pat. No. 5,371,684 (Attorney Docket No. SP041/1397.0190000), the disclosure of which is incorporated herein by reference.

Because it is possible that an instruction might get one of its inputs from a register that was written to by several other instructions, the present invention must choose which one is the real dependency. For example, if instructions 2 and 5 write to register 4 and instruction 7 reads register 4, then instruction 7 has two possible dependencies. In this case, it is assumed that since instruction 5 came after instruction 2

in the program, the programmer intended instruction 7 to use instruction 5's result and not instruction 2's. So, if an instruction can be dependent on several previous instructions, RRC 112 will consider it to be dependent on the highest numbered previous instruction.

Once TAL 122 has determined where the real dependencies are, it must locate the inputs for each instruction. In a preferred embodiment of the present invention, the inputs can come from the actual register file or an array temporary buffers 116. RRC 112 assumes that if an instruction has no dependencies, its inputs are all in the register file. In this case, RRC 112 passes the IXS1, IXS2 and IXS/D addresses that came from IFIFO 102 to the register file. If an instruction has a dependency, then RRC 112 assumes that the data is in temporary buffers 116. Since RRC 112 knows which previous instruction each instruction depends on, and since each instruction always writes to the same place in temporary buffers 116, RRC 112 can determine where in temporary buffers 116 an instruction's inputs are stored. It sends these addresses to register file read ports 119 and register file 117 outputs the data from temporary buffers 116 so that the instruction can use it.

The following is an example of tag assignments:

0: add r0, r1, r2

1: add r0, r2, r3

2: add r4, r5, r2

3: add r2, r3, r4

The following are the dependencies for the above operations (dependencies are represented by the symbol "#"):

I1S2#I0S/D

I3S1#I0S/D

I3S1#I2S/D

I3S2#I1S/D

First, look at I0; since it has no dependencies, its tags are equal to its original source register addresses:

I0S1 TAG=I0S1=r0

I0S2 TAG=I0S2=r1

I0S/D TAG=I0S/D=r2

I1 has one dependency, and its tags are as follows:

I1S1 TAG=I1S1=r0

I1S2 TAG=I0S/D=t0

where: (t0=inst. 0's slot in temporary buffer)

I1S/D TAG=I1S/D=r3

I2 is also independent:

I2S1 TAG=I2S1=r4

I2S2 TAG=I2S2=r5

I2S/D TAG=I2S/D=r2

I3S1 has two possible dependencies, I0S/D and I2S/D. Because TAL 122 must pick the last one (highest numbered one), I2S/D is chosen.

I3S1 TAG=I2S/D=t2

I3S2 TAG=I1S/D=t1

I3S/D TAG=I3S/D=r4

These tags are then sent to RPM 124 via bus 126 to be selected by Issuer 118. At the same time TAL 122 is preparing the tags, it is also monitoring the outputs of DCL 130 and passing them on to Issuer 118 using bus 120. TAL 122 chooses the proper outputs of DCL's 130 to pass to Issuer 118 by the same method that it chooses the tags that it sends to RPM 124.

Continuing the example, TAL 122 sends the following ready signals to Issuer 118:

I0S1 INFO=1

(Inst 0 is independent so it can start immediately)

I0S2 INFO=1

I0S/D INFO=1

I1S1 INFO=1

I1S2 INFO=DONE[0]

(DONE[0]=1 when I0 is done)

I1S/D INFO=1

I2S1 INFO=1

I2S2 INFO=1

I2S/D INFO=1

I3S1 INFO=DONE[2]

I3S2 INFO=DONE[1]

I3S/D READ=1

(The DONE signals come from DCL 130 via a bus 132. In connection with the present invention, the term "done" means the result of the instruction is in a temporary buffer or otherwise available at the output of a functional unit Contrastingly, the term "terminate" means the result of the instruction is in the register file.)

Turning now to FIG. 3, a representative block diagram of TAL 122 will be discussed. TAL 122 comprises 8 tag assignment logic blocks 302. Each TAL block 302 receives the corresponding data dependency results via buses 114, as well as further signals that come from the computer's Instruction Decode and control logic (not shown). The BKT bit signal forms the least significant bit of the tag. DONE[X] flags are for instructions 0 through 6, and indicate if instruction X is done. DBLREG[X] flags indicates which, if any, of the instructions is a double (long) word. Each TAL block 302 also receives its own instructions register addresses as inputs. The Misc. signals, DBLREG and BKT signals are all implementation dependent control signals. Each TAL block

302 outputs 3 TAGs 126 labeled IXS1, IXS2 and IXS/D, which are 6 bits. TAL 122 outputs the least significant 5 bits of each TAG signal to RPMs 124 and the most significant TAG to Issuer 118.

Each block 302 of FIG. 3 comprises three Priority Encoders (PE), one for S1, one for S2 and one for S/D. There is one exception however. I0 requires no tag assignment. Its tags are the same as the original S1, S2 and S/D addresses, because I0 is always independent.

An illustrative PE is shown in FIG. 9. PE 902 has eight inputs 904 and eight outputs 906. Inputs 904 for PE 902 are outputs 114 from DDC 108 which show where dependencies exist. For example, in the case of source register 1 (S1), I7S1 tag assign PE 902's seven inputs are the seven outputs 114 of DDC 108 that indicate whether I7S1 is dependent on I6D, whether I7S1 is dependent on I5D, and so on down to whether I7S1 is dependent on I0D. An eighth input, shown at reference number 908, is always tied high because there should always be an output from PE 902.

As stated before, if an instruction depends on several previous instructions, PE 902 will select and output only the most previous instruction (in program order) on which there is a dependency. This is accomplished by connecting the signal showing if there is a dependency on the most previous instruction to the highest priority input of the PE 902 and the signal showing if there is a dependency on the second most previous instruction to the input of PE 902 with the second highest priority and so on for all previous instructions. The input of the PE 902 with the lowest priority is always tied high so that at least one of PE 902's outputs will be asserted.

Outputs 906 are used as select lines for a MUX 910. MUX 910 has eight inputs 912 to which the tags for each instruction are applied.

To illustrate this, assume that I7 depends on I6 and I5, then, since I6 has a higher priority than I5, the bit corresponding to I6 at outputs 906 of PE 902 will be high. At the corresponding input 912 of MUX 910 will be I6's tag for S1 (recall PE 902 is for I7S1). Because I7 is dependent on I6, the location of I6's result must be output from MUX 910 so that it can be used by I7. I6's tag will therefore be selected and output on an output line 914. I6's done flag, DONE[6] must also be output from MUX 910 so that Issuer 118 will know when I7's input is ready. This data is passed to Issuer 118 via bus 120. Since an instruction can have up to three sources, TAL 122 monitors up to three dependencies for each instruction and sends three vectors for each instruction (totalling 24 vectors) to Issuer 118. If an instruction is independent, TAL 122 signals to Issuer 118 that the instruction can begin immediately.

The MSB of the tag outputs which are sent to RPMs 124 is used to indicate if the address is a register file address or a temporary buffer address. If an instruction is independent, then the five LSB outputs indicate the source register address. For instructions that have dependencies: the second MSB indicates that the address is for a 64 bit valve; the third through fifth MSB outputs specify the temporary buffer address; and the LSB output indicates which bucket is the current bucket, which is equal to the BKT signal in TAL 122.

Like DDC 108, TAL 122 has numerous implementation dependent, (i.e., special cases) that it handles. First, in an embodiment of the present invention, register number 0 of the register file is always equal to 0. Therefore, even if one instruction writes to register 0 and another reads from register 0, there will be no dependency between them. TAL 122 receives three signals from Instruction Decode Logic (IDL; not shown) for each instruction to indicate if one of that instruction's sources is register 0. If any of those is asserted, TAL 122 will ignore any dependencies for that particular input of that instruction.

Another special case occurs because under some circumstances, an instruction in bucket 0 will be guaranteed to not have any of the instructions in bucket 1 dependent on it. A four bit signal called BKT1_NODEP_ is sent to RRC 112 from the IEU control logic (not shown) and if BKT1_NODEP[X]=1 then RRC 112 knows to ignore any dependencies between instructions, 4,5,6 or 7 and instruction X.

An example for TAG assignment of instruction 7's source 1 (I7S1) is shown in a flowchart in FIGS. 6A–6B. TAL 122 first determines whether I7S1 is register 0, as shown at a conditional block 602. If the first source operand for I7 is register 0, the TAG is set equal to zero, and the I7S1's INFO flag is set equal to one, as shown in a block 604. If the first source operand (S1) for I7 is not register 0, TAL 122 then determines if I7S1 is dependent on I6S/D, as shown at a conditional block 606. If I7S1 is dependent on I6S/D flow then proceeds to a block 610 where I7S1's TAG is set equal to {1,DBLREG[6],0,1,0,BKT} and I7S1's INFO flag is set equal to DONE[6], as shown at a block 610. If either of the condition tested at a conditional block 606 is not met, flow proceeds to conditional block 612 where TAL 122 determines if I7S1 is dependent on I58/D. If there is a dependency, flow then proceeds to block 616 where TAL 122 sets I7S1's TAG equal to {1,DBLREG[5],0,0,1,BKT} and I7S1's INFO flag is set equal to DONE[5]. If the condition tested at block 612 is not met, flow proceeds to a block 618 where TAL 122 determines if I7S1 is dependent on I4S/D.

As evident by inspection of the remaining sections of FIGS. 6A and 6B, similar TAG determinations are made depending on whether I7S1 is dependent on I4S/D, I3S/D, I2S/D, I1S/D and I0S/D, as shown at sections 620, 622, 624, 626 and 628, respectively. Finally, if instruction 7 is independent of instruction 0 or if all instructions in bucket 1 are independent of instruction 0 (i.e., if BKT1_NODEP[0]=1), as tested at a conditional block 630, the flow proceeds to block 632 where TAL 122 sets I7S1's TAG equal to {0,I7S1} and I7S1's INFO flag equal to 1. It should be noted for the above example that I7S1 TAG signals are forwarded directly the register file port MUXes of register file 117. The I7S1 INFO signals are sent to Issuer 118 to tell it when I7's S1 input is ready.

A representative block diagram of Issuer 118 is shown in FIG. 10. In a preferred embodiment, Issuer 118 has one scanner block 1002 for each resource (functional unit) that has to be allocated. In this example, Issuer 118 has scanner blocks FU1, FU2, FU3, FU4 through FUn. Requests for functional units are generated from instruction information by decoding logic (not shown) in a known manner, which are sent to scanners 1002 via bus 123. Each scanner block 1002 scans from instruction I0 to I7 and selects the first request for the corresponding functional unit to be serviced during that cycle.

In the case of multiple register files (integer, floating and/or boolean), Issuer 118 is capable of issuing instructions having operands stored in different register files. For example, an ADD instruction may have a first operand from the floating point register file and a second operand from the integer register file. Instructions with operands from different register files are typically given higher issue priority (i.e., they are issued first). This issuing technique conserves processor execution time and functional unit resources.

In a further embodiment in which IEU 100 may include two ALU's, ALU scanning becomes a bit more complicated. For speed reasons, one ALU scanner block scans from I0 to

I7, while the other scanner block scans from I7 to I0. This is how two ALU requests are selected. With this scheme it is possible that an ALU instruction in bucket 1 will get issued before an ALU instruction in bucket 0, while increasing scanning efficiency.

Scanner outputs 1003 are selected by MUXing logic 1004. A set of SELect inputs 1006 for MUX 1004 receive three 8-bit vectors (one for each operand) from TAL 122 via bus 120. The vectors indicate which of the eight instructions have no dependencies and are ready to be issued. Issuer 118 must wait for this information before it can start to issue any instructions. Issuer 118 monitors these vectors and when all three go high for a particular instruction, Issuer 118 knows that the inputs for that instruction are ready. Once the necessary functional unit is ready, the issuer can issue that instruction and send select signals to the register file port MUXes to pass the corresponding instructions outputs to register file 117.

In a preferred embodiment of the present invention, after Issuer 118 is done it provides two 8-bit vectors per register file back to RRC 112 via MUXOUTputs 1008 to bus 121. These vectors indicate which instructions are issued this cycle, are used a select lines for RPMs 124.

The maximum number of instructions that can be issued simultaneously for each register file is restricted by the number of register file read ports available. A data dependency with a previous uncompleted instruction may prevent an instruction from being issued. In addition, an instruction may be prevented from being issued if the necessary functional unit is allocated to another instruction.

Several instructions, such as load immediate instructions, Boolean operations and relative conditional branches, may be issued independently, because they may not require resources other than register file read ports or they may potentially have no dependencies.

The last section of RRC 112 is the register file port MUX (RPM) section 124. The function of RPMs 124 is to provide a way for Issuer 118 to get data out of register files 117 for each instruction to use. RPMs 124 receive tag information via bus 126, and the select lines for RPMs 124 come from Issuer 118 via a bus 121 and also from the computer's IEU control logic. The selected TAGs comprise read addresses that are sent to a predetermined set of ports 119 of register file 117 using bus 128.

The number and design of RPMs 124 depend on the number of register files and the number of ports on each register file. One embodiment of RPMs 124 is shown in FIG. 4. In this embodiment, RPMs 124 comprises 3 register port file MUXes 402, 404 and 406. MUX 402 receives as inputs the TAGs of instructions 0–7 corresponding to the source register field S1 that are generated by TAL 122. MUX 404 receives as inputs the TAGs of instructions 0–7 corresponding to the source register field S2 that are generated by TAL 122. MUX 406 receives as inputs the TAGs of instructions 0–7 corresponding to the source/destination register field S/D that are generated by TAL 122. The outputs of MUXes 402, 404 and 406 are connected to the read addresses ports of register file 117 via bus 128.

RRC 112 and Issuer 118 allow the processor to execute instructions simultaneously and out of program order. An IEU for use with the present invention is disclosed in commonly owned, co-pending application Ser. No. 07/817,810 (Attorney Docket No. SP015/1397.0280001), the disclosure of which is incorporated herein by reference.

While various embodiments of the present invention have been described above, it should be understood that they have been presented by way of example, and not limitation. Thus the breadth and scope of the present invention should not be limited by any of the above-described exemplary embodiments, but should be defined only in accordance with the following claims and their equivalents.

What is claimed is:

1. A system for register renaming in a computer system capable of out-of-order instruction execution, comprising:

a temporary buffer comprising a plurality of storage locations for storing execution results, wherein an execution result for an instruction is stored at one of said plurality of storage locations, said one of a plurality of storage locations being determined by a location of said instruction in an instruction window;

tag assignment logic for receiving data dependency results from a data dependency checker and for outputting a tag in place of a register address for an operand of a first instruction if said first instruction is dependent on a previous one of said plurality of instructions in said instruction window for said operand, wherein said tag represents an address of said operand in one of said plurality of storage locations.

2. The register renaming system of claim 1, further comprising means for transferring the execution results in said plurality of storage locations in said temporary buffer to register file locations in-order based on the order of instructions in said instruction window.

3. The register renaming system of claim 2, wherein said means for transferring transfers a group of execution results from said temporary buffer to said register file simultaneously.

4. The register renaming system of claim 3, wherein said means for transferring transfers an execution result for an instruction from said temporary buffer to said register file when all execution results for all prior instructions are retirable.

5. The register renaming system of claim 1, wherein said tags comprise an address and a 1-bit identifier that indicates whether said address within said tags is an address within said register file locations or said temporary buffer locations.

6. The register renaming system of claim 1, further comprising means for passing said tags to read address ports of said temporary buffer for accessing said instruction execution results.

7. A computer system, comprising:

a memory unit for storing program instructions;

a bus coupled to said memory unit for retrieving said program instructions; and

a processor coupled to said bus, wherein said processor comprises a register renaming system, comprising:

a temporary buffer comprising a plurality of storage locations for storing execution results, wherein an execution result for an instruction is stored at one of said plurality of storage locations, said one of a plurality of storage locations being determined by a location of said instruction in an instruction window;

tag assignment means for receiving data dependency results from a data dependency checker and for outputting a tag in place of a register address for an operand of a first instruction if said first instruction is dependent on a previous one of said plurality of instructions in said instruction window for said operand, wherein said tag represents an address of said operand in one of said plurality of storage locations.

8. The computer system of claim 7, wherein said processor further comprises means for transferring the execution

results in said plurality of storage locations in said temporary buffer to register file locations in-order based on the order of instructions in said instruction window.

9. The computer system of claim 8, wherein said means for transferring transfers a group of execution results from said temporary buffer to said register file simultaneously.

10. The computer system of claim 9, wherein said means for transferring transfers an execution result for an instruction from said temporary buffer to said register file when all execution results for all prior instructions are retirable.

11. The computer system of claim 7, wherein said tags comprise an address and a 1-bit identifier that indicates whether said address within said tags is an address within said register file locations or said temporary buffer locations.

12. The computer system of claim 7, wherein said processor further comprises means for passing said tags to read address ports of said temporary buffer for accessing said execution results.

13. A register renaming method, comprising the steps of:

(1) storing, in a temporary buffer, out-of-order execution results in storage locations determined by the location of an instruction in an instruction window;

(2) generating at least one tag to specify an address in said temporary buffer at which said out-of-order execution results are temporarily stored; and

(3) outputting a tag in place of a register address for an operand of a first instruction if a data dependency result indicates that said first instruction is dependent on a previous instruction in said instruction window, wherein said tag comprises an address of said operand in said temporary buffer.

14. The register renaming method of claim 13, further comprising the step of transferring said out-of-order execution results in said temporary buffer to a register file in-order based on the order of instructions in said instruction window.

15. The register renaming method of claim 14, further comprising the step of transferring a group of execution results from said temporary buffer to said register file simultaneously.

16. The register renaming method of claim 15, further comprising the step of transferring an out-of-order execution result from said temporary buffer to said register file when all execution results for all prior instructions are retirable.

17. The register renaming method of claim 13, further comprising the step of determining data dependencies between the instructions in said instruction window to produce said data dependency results.

18. The register renaming method of claim 13, wherein said step (2) further comprises the step of generating tags that comprise an address and a 1-bit identifier that indicates whether said address within said tags is an address within a register file or said temporary buffer.

19. The register renaming method of claim 13, further comprising the step of passing said tags to read address ports of said temporary buffer for accessing said out-of-order execution results.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 5,737,624
DATED : April 7, 1998
INVENTOR(S) : Garg et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Title page,
Item [63], **Related U.S. Application Data**, please replace "March 21, 1992" with -- March 31, 1992 --.

Drawings,
Sheets 6 and 7, please replace Figures 6A and 6B with the attached Figures 6A and 6B.

Column 1,
Line 10, please replace "March 21, 1992" with -- March 31, 1992 --.

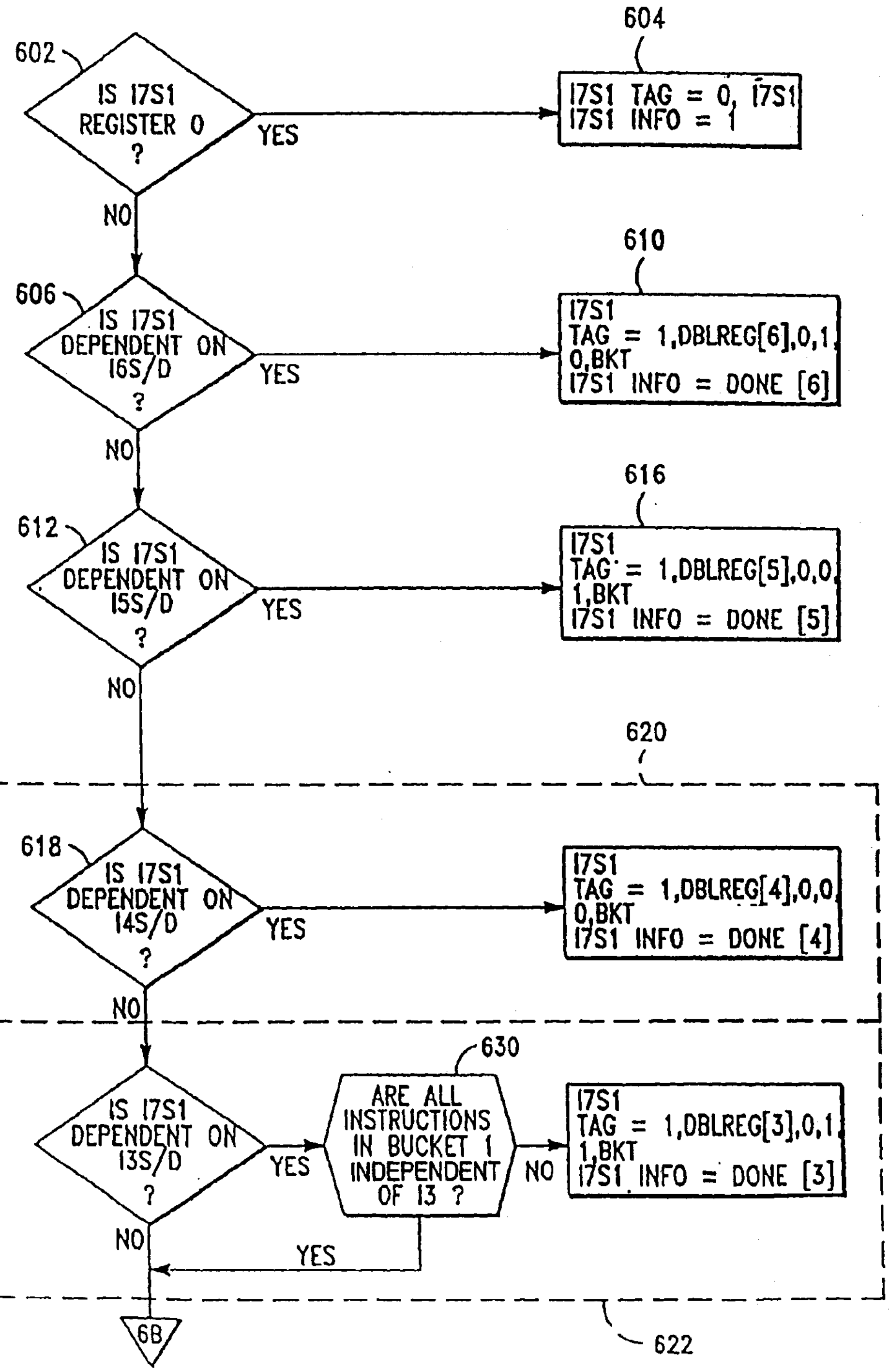
Column 5,
Line 3, please replace "if" with -- If --.

Column 12,
Line 56, please replace "derailed" with -- detailed --.

Signed and Sealed this

Second Day of December, 2003

JAMES E. ROGAN
Director of the United States Patent and Trademark Office



602
IS 17S1 REGISTER 0 ?
YES
604
17S1 TAG = 0, 17S1
17S1 INFO = 1
NO
606
IS 17S1 DEPENDENT ON 16S/D ?
YES
610
17S1
TAG = 1,DBLREG[6],0,1,
0,BKT
17S1 INFO = DONE [6]
NO
612
IS 17S1 DEPENDENT ON 15S/D ?
YES
616
17S1
TAG = 1,DBLREG[5],0,0,
1,BKT
17S1 INFO = DONE [5]
NO
620
618
IS 17S1 DEPENDENT ON 14S/D ?
YES
17S1
TAG = 1,DBLREG[4],0,0,
0,BKT
17S1 INFO = DONE [4]
NO
IS 17S1 DEPENDENT ON 13S/D ?
YES
630
ARE ALL INSTRUCTIONS IN BUCKET 1 INDEPENDENT OF 13 ?
NO
17S1
TAG = 1,DBLREG[3],0,1,
1,BKT
17S1 INFO = DONE [3]
NO
YES
6B
622

FIG.—6A

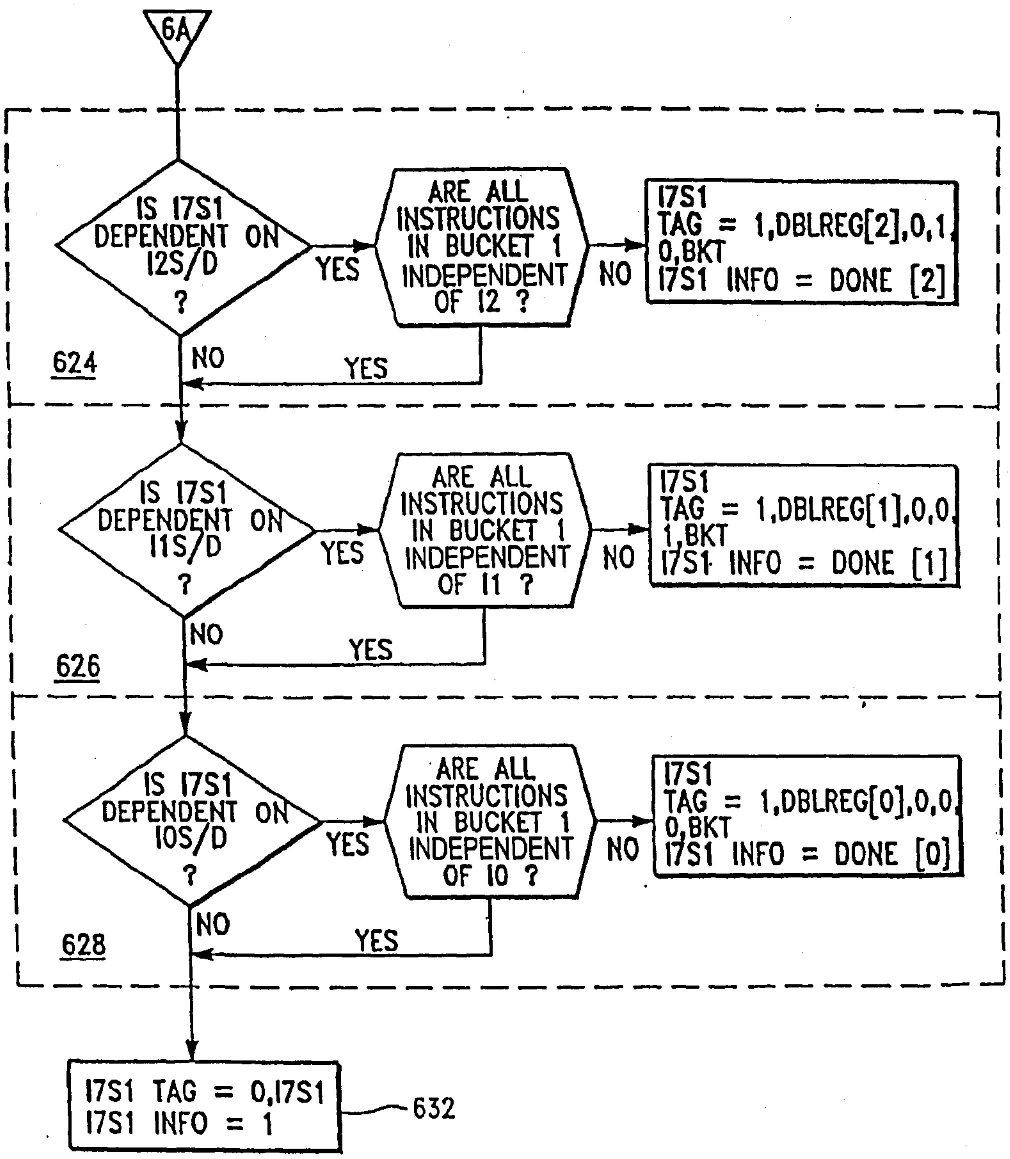

6A
IS I7S1 DEPENDENT ON I2S/D ?
YES
ARE ALL INSTRUCTIONS IN BUCKET 1 INDEPENDENT OF I2 ?
NO
I7S1 TAG = 1,DBLREG[2],0,1, 0,BKT
I7S1 INFO = DONE [2]
624
NO
YES
IS I7S1 DEPENDENT ON I1S/D ?
YES
ARE ALL INSTRUCTIONS IN BUCKET 1 INDEPENDENT OF I1 ?
NO
I7S1 TAG = 1,DBLREG[1],0,0, 1,BKT
I7S1 INFO = DONE [1]
626
NO
YES
IS I7S1 DEPENDENT ON I0S/D ?
YES
ARE ALL INSTRUCTIONS IN BUCKET 1 INDEPENDENT OF I0 ?
NO
I7S1 TAG = 1,DBLREG[0],0,0, 0,BKT
I7S1 INFO = DONE [0]
628
NO
YES
I7S1 TAG = 0,I7S1
I7S1 INFO = 1
632

FIG.—6B

EXHIBIT I

US005974526A

United States Patent [19]

Garg et al.

[11] **Patent Number: 5,974,526**

[45] **Date of Patent: *Oct. 26, 1999**

[54] **SUPERSCALAR RISC INSTRUCTION SCHEDULING**

[75] Inventors: **Sanjiv Garg**, Freemont; **Kevin Ray Iadonato**, San Jose; **Le Trong Nguyen**, Sereno; **Johannes Wang**, Redwood City, all of Calif.

[73] Assignee: **Seiko Corporation**, Tokyo, Japan

[*] Notice: This patent is subject to a terminal disclaimer.

[21] Appl. No.: **08/990,414**

[22] Filed: **Dec. 15, 1997**

Related U.S. Application Data

[63] Continuation of application No. 08/594,401, Jan. 31, 1996, Pat. No. 5,737,624, which is a continuation of application No. 08/219,425, Mar. 29, 1994, Pat. No. 5,497,499, which is a continuation of application No. 07/860,719, Mar. 31, 1992, abandoned.

[51] **Int. Cl.**[6] **G06F 9/38**

[52] **U.S. Cl.** **712/23**; 712/217; 712/218; 712/215

[58] **Field of Search** 395/800.23, 391, 395/393, 394; 712/23, 206, 207, 217, 215, 218, 219, 216

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,901,233	2/1990	Liptay	395/375
4,903,196	2/1990	Pomerene et al.	364/200
4,942,525	7/1990	Shintani et al.	395/375
4,992,938	2/1991	Cocke et al.	364/200
5,067,069	11/1991	Fite et al.	395/375

(List continued on next page.)

FOREIGN PATENT DOCUMENTS

0 378 195 A3	7/1990	European Pat. Off. .
0 515166	11/1992	European Pat. Off. .
0 533337	3/1993	European Pat. Off. .
WO 91/20031	12/1991	WIPO .

OTHER PUBLICATIONS

"Critical Issues Regarding HPS, A High Performance Microarchitecture", Yale N. Patt, Stephen W. Melvin, Wen–Mei Hwu and Michael C. Shebanow; The 18th Annual Workshop on Microprogramming, Pacific Grove, California, Dec. 3–6, 1985, IEEE Computer Order No. 653, pp. 109–116.

"HPS, A New Microarchitecture: Rationale and Introduction", Yale N. Patt, Wen–Mei Hwu and Michael C. Shebanow; The 18th Annual Workshop on Microprogramming, Pacific Grove, California, Dec. 3–6, 1985; IEEE Computer Society Order No. 653, pp. 103–108.

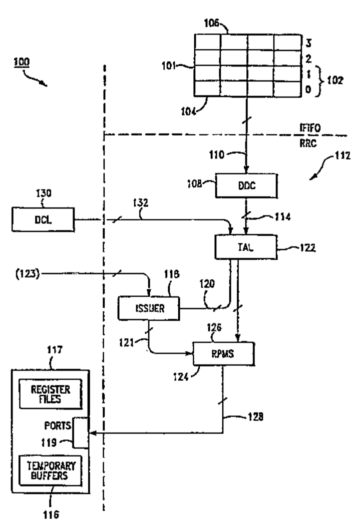
(List continued on next page.)

Primary Examiner—Larry D. Donaghue
Attorney, Agent, or Firm—Sterne, Kessler, Goldstein & Fox P.L.L.C.

[57] **ABSTRACT**

A register renaming system for out-of-order execution of a set of reduced instruction set computer instructions having addressable source and destination register fields, adapted for use in a computer having an instruction execution unit with a register file accessed by read address ports and for storing instruction operands. A data dependance check circuit is included for determining data dependencies between the instructions. A tag assignment circuit generates one of more tags to specify the location of operands, based on the data dependencies determined by the data dependance check circuit. A set of register file port multiplexers select the tags generated by the tag assignment circuit and pass the tags onto the read address ports of the register file for storing execution results.

34 Claims, 9 Drawing Sheets



U.S. PATENT DOCUMENTS

5,109,495	4/1992	Fite et al.	395/375
5,142,633	8/1992	Murray et al.	395/375
5,214,763	5/1993	Blaner et al.	395/375
5,222,244	6/1993	Carbine et al.	395/800
5,226,126	7/1993	McFarland et al.	395/375
5,251,306	10/1993	Tran	395/375
5,261,071	11/1993	Lyon	395/425
5,345,569	9/1994	Tran	395/375
5,355,457	10/1994	Shebanow et al.	395/375
5,398,330	3/1995	Johnson	395/375
5,448,705	9/1995	Nguyen et al.	395/375
5,487,156	1/1996	Popescu et al.	395/375
5,497,499	3/1996	Garg et al.	395/800.23
5,561,776	10/1996	Popescu et al.	395/375
5,574,927	11/1996	Scantlin	395/800
5,592,636	1/1997	Popescu et al.	395/586
5,625,837	4/1997	Popescu et al.	395/800
5,627,983	5/1997	Popescu et al.	395/393
5,708,841	1/1998	Popescu et al.	355/800
5,737,624	4/1998	Garg et al.	395/800.23
5,778,210	7/1998	Henstrom et al.	395/394
5,797,025	8/1998	Popescu et al.	395/800
5,832,205	11/1998	Kelly et al.	395/185.06
5,832,293	11/1998	Popescu et al.	395/800.23

OTHER PUBLICATIONS

Peleg et al., "Future Trends in Microprocessors: Out-Of-Order Execution, Spec. Branching and Their CISC Performance Potential", Mar. 1991.

Lightner et al., "The Metaflow Architecture", pp. 11–12, 63–68, IEEE Micro Magazine, Jun. 1991.

John L. Hennessy & David A Patterson, *Computer Architecture A Quantitative Approach,* Ch. 6.4, 6.7 and p. 449, 1990.

Hwu, Wen-mei, Steve Melvin, Mike Shebanow, Chein Chen, Jia-juin Wei, Yale Patt, "An HPS Implementation of VAX: Initial Design and Analysis", Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences, pp. 282–291, 1986.

Hwu et al., "Experiments with HPS, a Restricted Data Flow Microarchitecture for High Performance Computers", *COMPCON 86,* 1986.

Hwu, Wen-mei and Yale N. Patt, "HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality", Proceedings of the 18th International Symposium on Computer Architecture, pp. 297–306, Jun. 1986.

Yale N. Patt, Stephen W. Melvin, Wen-mei Hwu, Michael C. Shebanow, Chein Chen, Jiajuin Wei, "Run-Time Generation of HPS Microinstructions From a Vax Instruction Stream", Proceedings of Micro 19 Workshop, New York, New York, pp. 1–7, Oct., 1986.

Swenson, John A. and Yale N. Patt, "Heirarchical Registers for Scientific Computers", *St. Malo '88,* University of California at Berkeley, pp. 346–353, 1988.

Butler, Michael and Yale Patt, "An Improved Area-Efficient Register Alias Table for Implementing HPS", University of Michigan, Ann Arbor, Michigan, pp. 1–15, Jan., 1990.

Uvieghara, G.A., W.Hwu, Y. Nakagome, D.K. Jeong, D. Lee, D.A. Hodges, Y. Patt, "An Experimental Single-Chip Data Flow CPU", Symposium on ULSI Circuits Design Digest of Technical Papers, May, 1990.

Melvin, Stephen and Yale Patt, "Exploiting Fine-Grained Parallelism Through a Combination of Hardware and Software Techniques", Proceedings From ISCA-18, pp. 287–296, May, 1990.

Butler, Michael, Tse-Yu Yeh, Yale Patt, Mitch Alsup, Hunter Scales and Michael Shebanow, "Single Instruction Stream Parallelism Is Greater Than Two" Proceedings of ISCA-18, pp. 276–286, May, 1990.

Uvieghara, Gregory A., Wen-mei, W. Hwu, Yoshinobu Nakagome, Deog-Kyoon Jeong, David D. Lee, David A. Hodges and Yale Patt, "An Experimental Single-Chip Data Flow CPU", *IEEE Journal of Solid-State Circuits,* vol. 27, No. 1, pp. 17–28, Jan., 1992.

Gee, Jeff, Stephen W. Melvin, Yale N. Patt, "The Implementationof Prolog via VAX 8600 Microcode", Proceedings of Micro 19, New York City, pp. 1–7, Oct., 1986.

Hwu, Wen-mei Hwu and Yale N. Patt, "Design Choices for the HPSm Microprocessor Chip", Proceedings of the Twentieth Annual Hawaii International Conference on System Sciences, pp. 330–336, 1987.

Wilson, James E., Steve Melvin, Michael Shebanow, Wen-mei Hwu and Yale N. Patt, "On Turning the Microarchitecture of an HPS Implementation of the VAX", Proceedings of Micro 20, pp. 162–167, Dec., 1987.

Hwu, Wen-mei and Yale N. Patt, "HPSm2: A Refined Single-chip Microengine", *HICSS '88,* pp. 30–40, 1988.

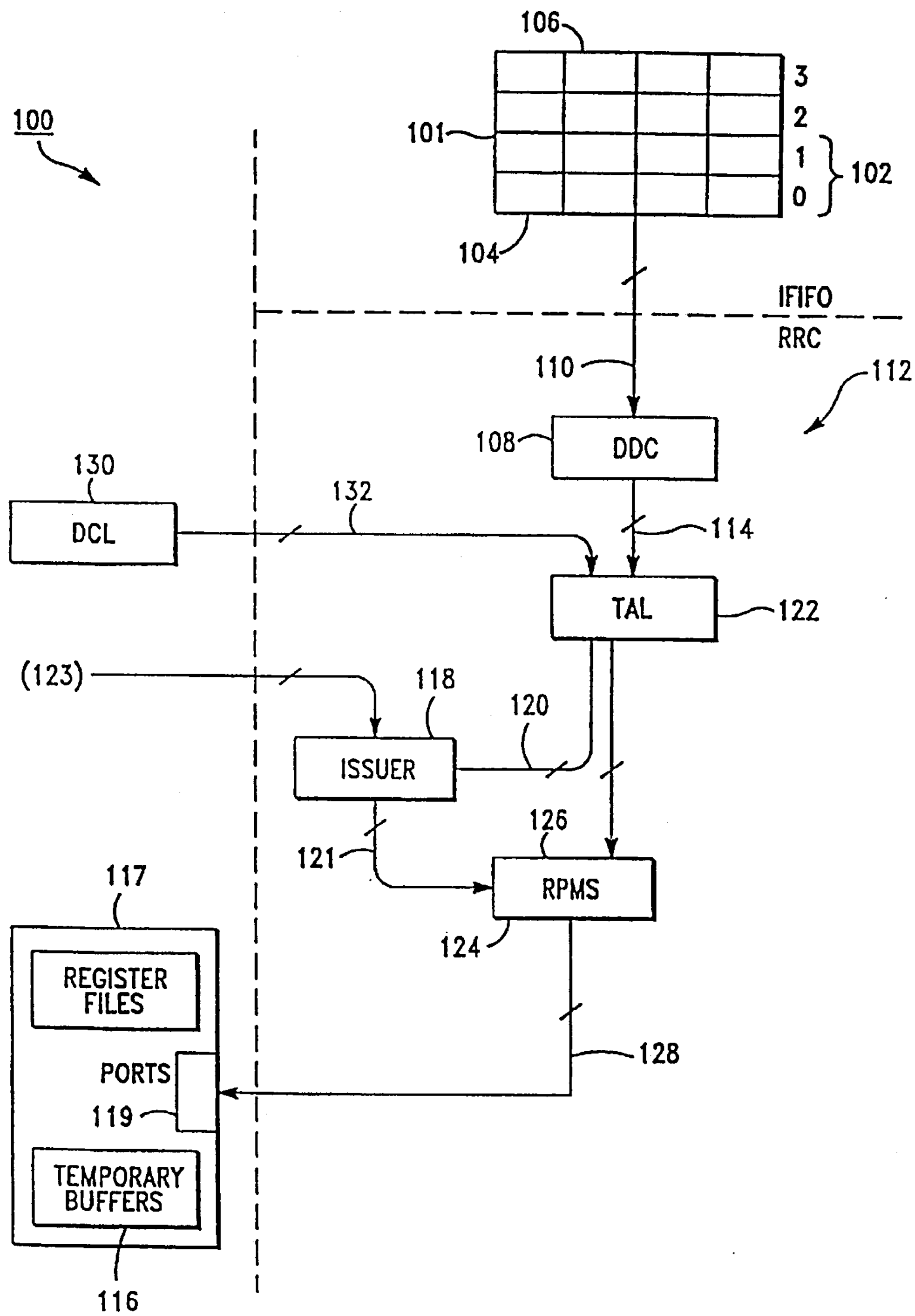
Keller, "Look-Ahead Processors"; Dec. 1975, pp. 177–194.

Dywer, A Multiple, Out-of-Order, Instruction Issuing System For SuperScaler Processors, (All); Aug. 1991.

Lightner et al., "The Metaflow Lightning" Chip Set Mar. 1991 *IEEE Lightning Outlined, Microprocessor Report* Sep. 1990.

Michael D. Smith et al., "Limits on Multiple Instruction Issue," *Computer Architecture News,* No. 2, Apr. 17, 1989, pp. 290–302.

Gurindar S. Sohi et al., "Instruction Issue Logic for High Performance, Interruptable Pipelined Processors," The 14th Annual International Symposium on Computer Architecture, Jun. 2–5, 1987, pp. 27–34.



100
106
101
3
2
1
0
102
104
IFIFO
RRC
110
112
108
DDC
130
DCL
132
114
TAL
122
(123)
118
120
ISSUER
126
121
RPMS
124
117
REGISTER FILES
PORTS
119
TEMPORARY BUFFERS
116
128

FIG.—1

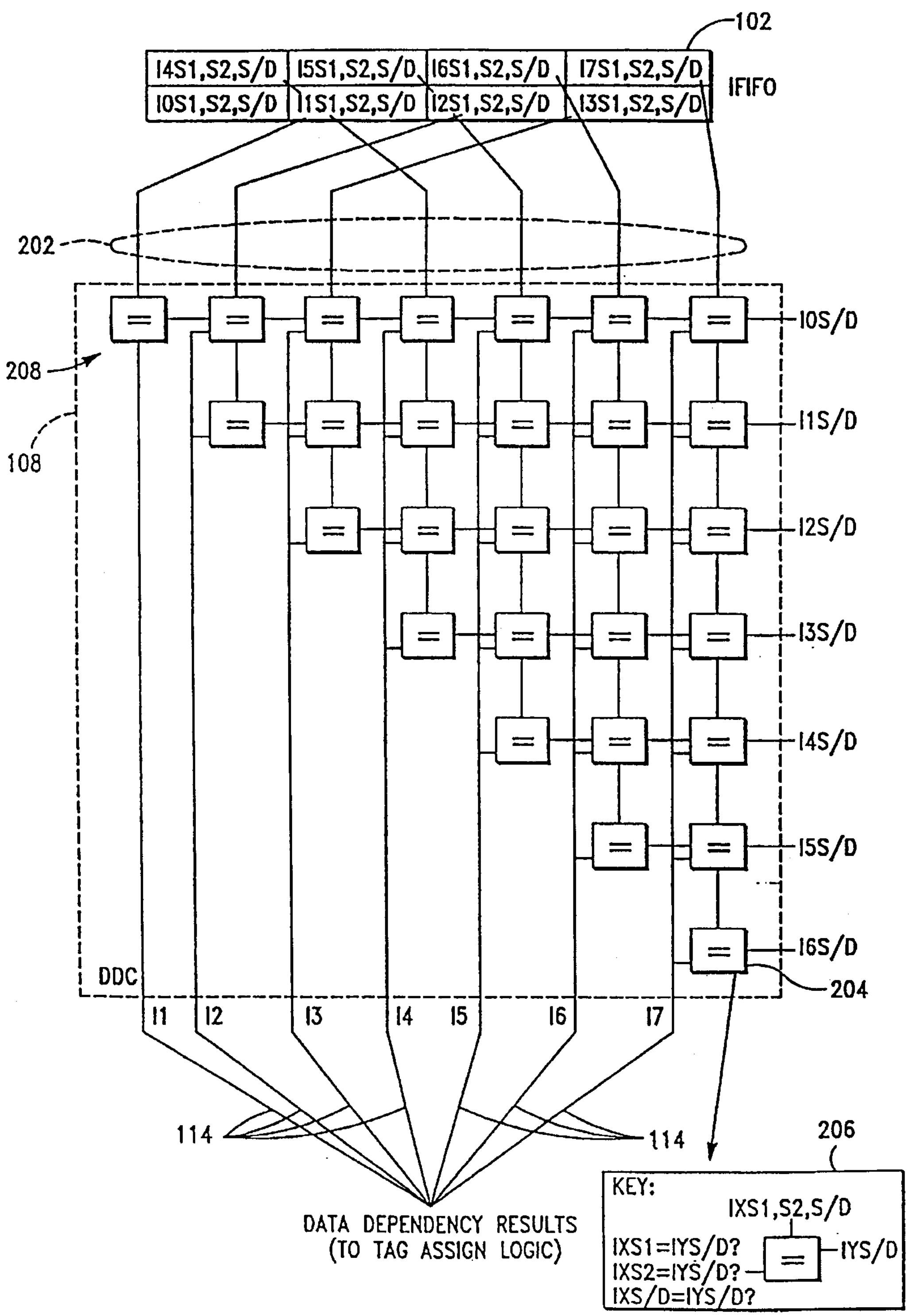

102
I4S1,S2,S/D
I5S1,S2,S/D
I6S1,S2,S/D
I7S1,S2,S/D
I0S1,S2,S/D
I1S1,S2,S/D
I2S1,S2,S/D
I3S1,S2,S/D
IFIFO
202
208
108
I0S/D
I1S/D
I2S/D
I3S/D
I4S/D
I5S/D
I6S/D
204
DDC
I1
I2
I3
I4
I5
I6
I7
114
114
206
DATA DEPENDENCY RESULTS
(TO TAG ASSIGN LOGIC)
KEY:
IXS1,S2,S/D
IXS1=IYS/D?
IXS2=IYS/D?
IXS/D=IYS/D?
IYS/D

FIG.—2

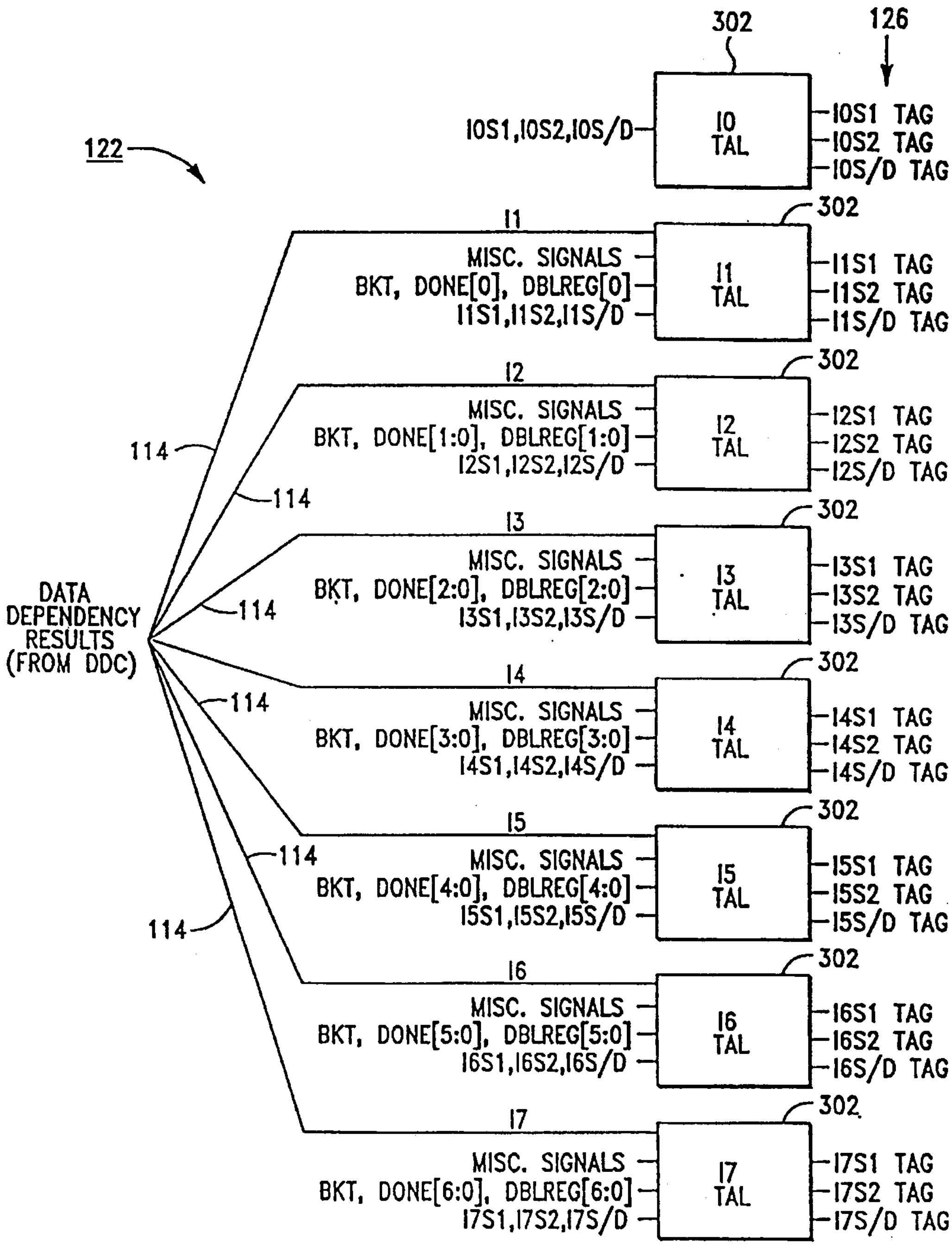

302
126
122
I0S1,I0S2,I0S/D
I0
TAL
I0S1 TAG
I0S2 TAG
I0S/D TAG
I1
MISC. SIGNALS
BKT, DONE[0], DBLREG[0]
I1S1,I1S2,I1S/D
I1
TAL
I1S1 TAG
I1S2 TAG
I1S/D TAG
I2
MISC. SIGNALS
BKT, DONE[1:0], DBLREG[1:0]
I2S1,I2S2,I2S/D
I2
TAL
I2S1 TAG
I2S2 TAG
I2S/D TAG
I3
MISC. SIGNALS
BKT, DONE[2:0], DBLREG[2:0]
I3S1,I3S2,I3S/D
I3
TAL
I3S1 TAG
I3S2 TAG
I3S/D TAG
I4
MISC. SIGNALS
BKT, DONE[3:0], DBLREG[3:0]
I4S1,I4S2,I4S/D
I4
TAL
I4S1 TAG
I4S2 TAG
I4S/D TAG
I5
MISC. SIGNALS
BKT, DONE[4:0], DBLREG[4:0]
I5S1,I5S2,I5S/D
I5
TAL
I5S1 TAG
I5S2 TAG
I5S/D TAG
I6
MISC. SIGNALS
BKT, DONE[5:0], DBLREG[5:0]
I6S1,I6S2,I6S/D
I6
TAL
I6S1 TAG
I6S2 TAG
I6S/D TAG
I7
MISC. SIGNALS
BKT, DONE[6:0], DBLREG[6:0]
I7S1,I7S2,I7S/D
I7
TAL
I7S1 TAG
I7S2 TAG
I7S/D TAG
114
DATA DEPENDENCY RESULTS (FROM DDC)

FIG.—3

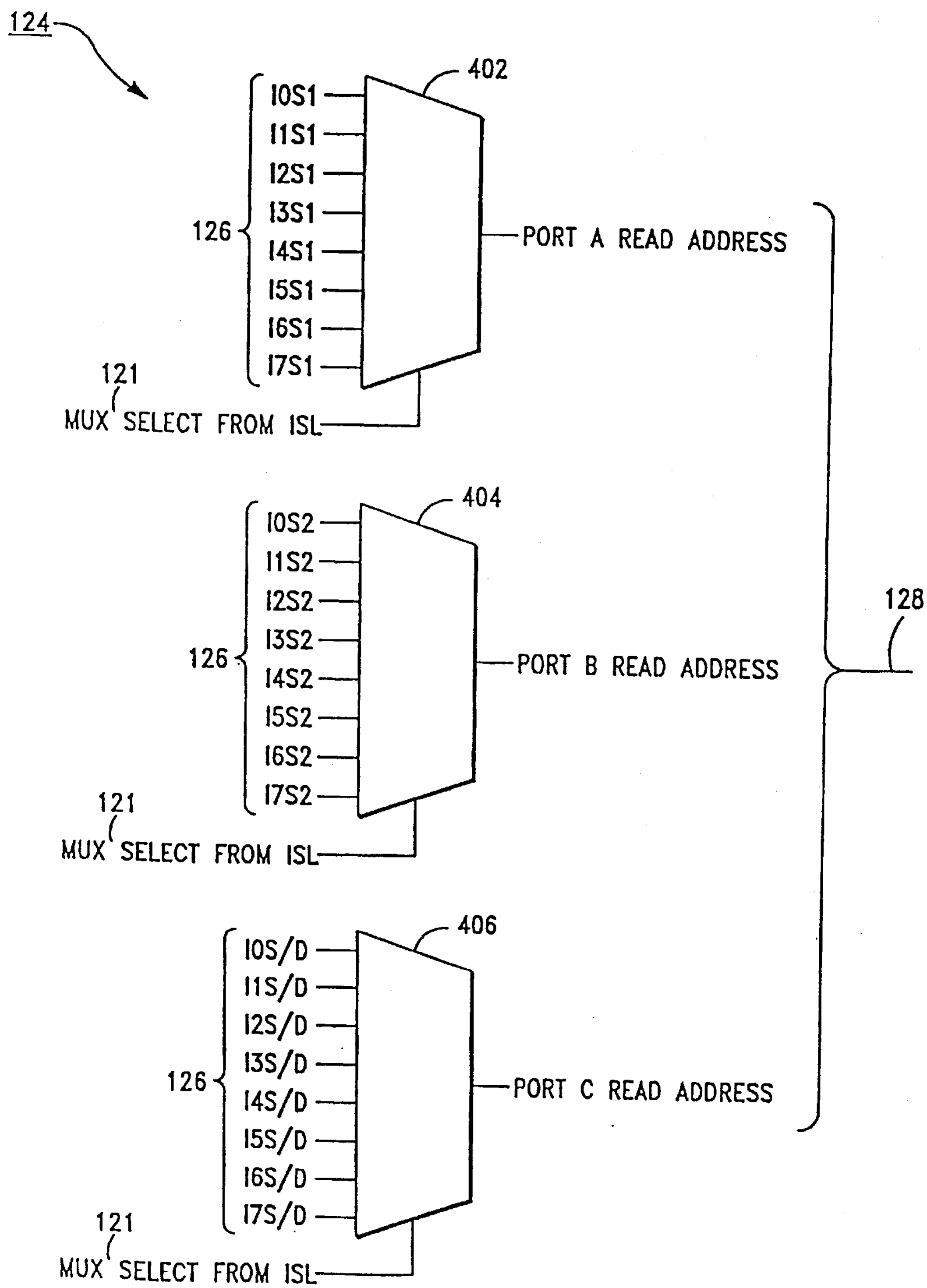

124
402
I0S1
I1S1
I2S1
I3S1
I4S1
I5S1
I6S1
I7S1
126
PORT A READ ADDRESS
121
MUX SELECT FROM ISL
404
I0S2
I1S2
I2S2
I3S2
I4S2
I5S2
I6S2
I7S2
126
PORT B READ ADDRESS
128
121
MUX SELECT FROM ISL
406
I0S/D
I1S/D
I2S/D
I3S/D
I4S/D
I5S/D
I6S/D
I7S/D
126
PORT C READ ADDRESS
121
MUX SELECT FROM ISL

FIG.—4

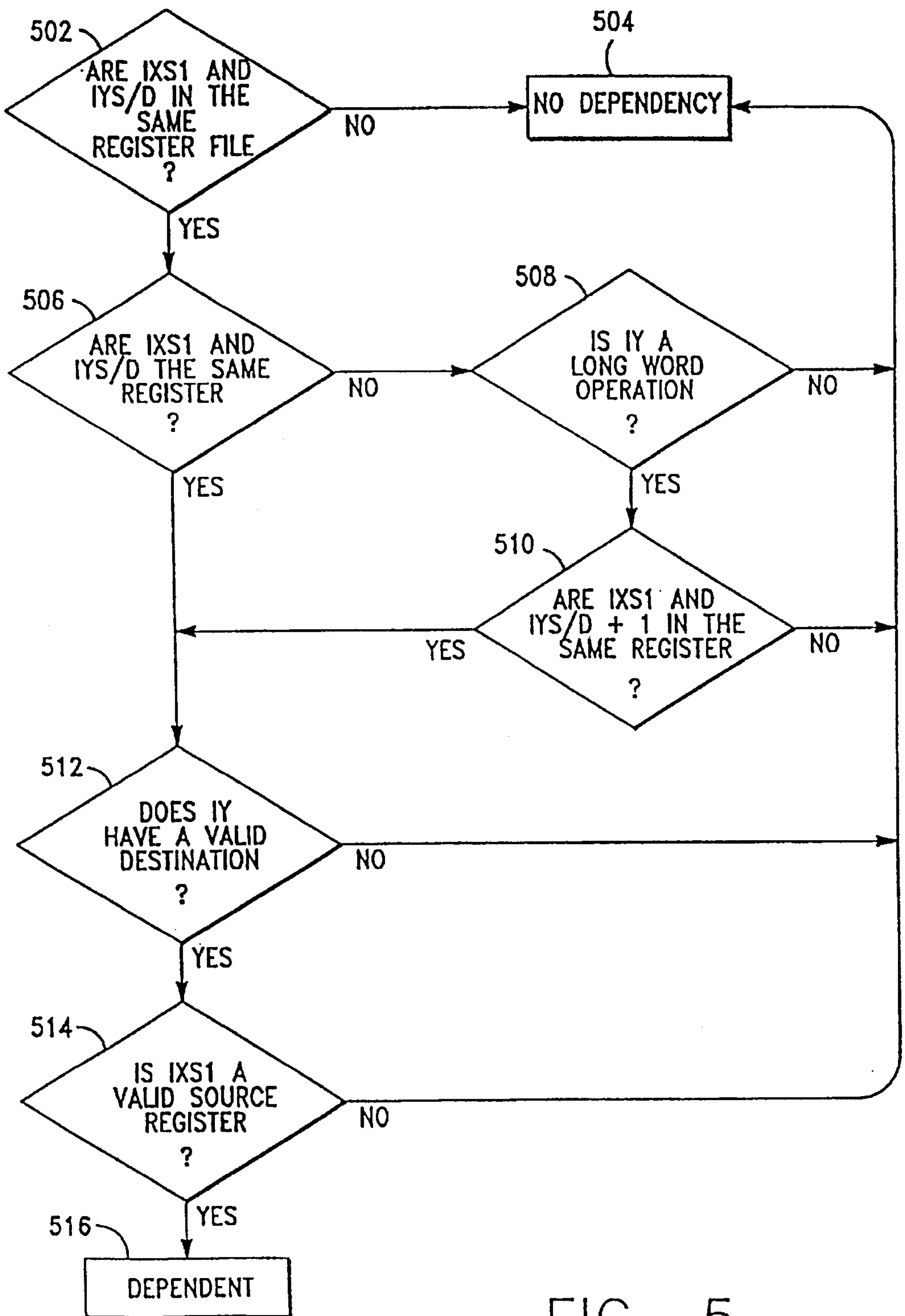

502
ARE IXS1 AND IYS/D IN THE SAME REGISTER FILE ?
NO
504
NO DEPENDENCY
YES
506
ARE IXS1 AND IYS/D THE SAME REGISTER ?
NO
508
IS IY A LONG WORD OPERATION ?
NO
YES
YES
510
ARE IXS1 AND IYS/D + 1 IN THE SAME REGISTER ?
YES
NO
512
DOES IY HAVE A VALID DESTINATION ?
NO
YES
514
IS IXS1 A VALID SOURCE REGISTER ?
NO
YES
516
DEPENDENT

FIG.—5

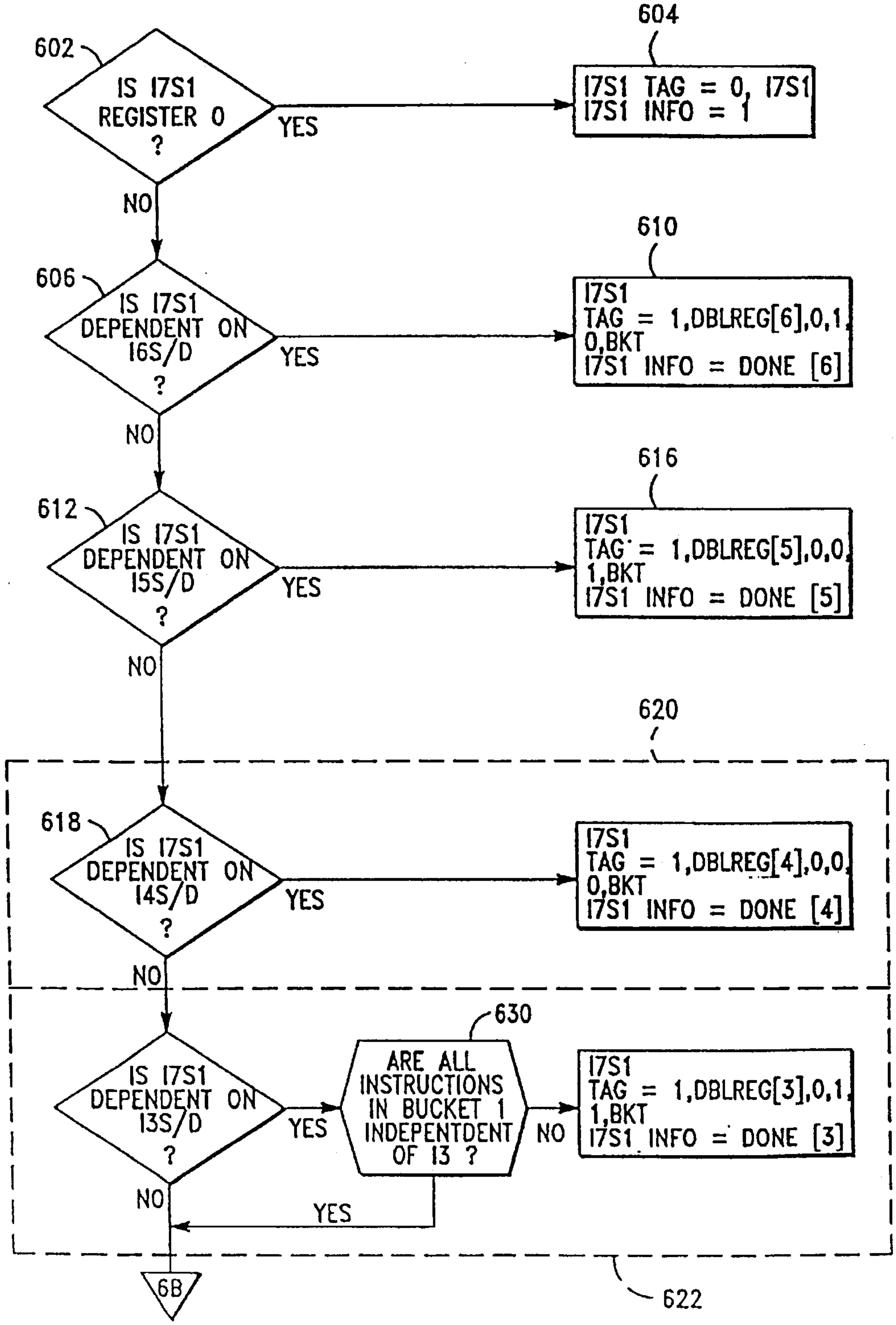

602
IS I7S1 REGISTER 0 ?
YES
604
I7S1 TAG = 0, I7S1
I7S1 INFO = 1
NO
606
IS I7S1 DEPENDENT ON I6S/D ?
YES
610
I7S1
TAG = 1,DBLREG[6],0,1,
0,BKT
I7S1 INFO = DONE [6]
NO
612
IS I7S1 DEPENDENT ON I5S/D ?
YES
616
I7S1
TAG = 1,DBLREG[5],0,0,
1,BKT
I7S1 INFO = DONE [5]
NO
620
618
IS I7S1 DEPENDENT ON I4S/D ?
YES
I7S1
TAG = 1,DBLREG[4],0,0,
0,BKT
I7S1 INFO = DONE [4]
NO
630
IS I7S1 DEPENDENT ON I3S/D ?
YES
ARE ALL INSTRUCTIONS IN BUCKET 1 INDEPENTDENT OF I3 ?
NO
I7S1
TAG = 1,DBLREG[3],0,1,
1,BKT
I7S1 INFO = DONE [3]
NO
YES
6B
622

FIG. — 6A

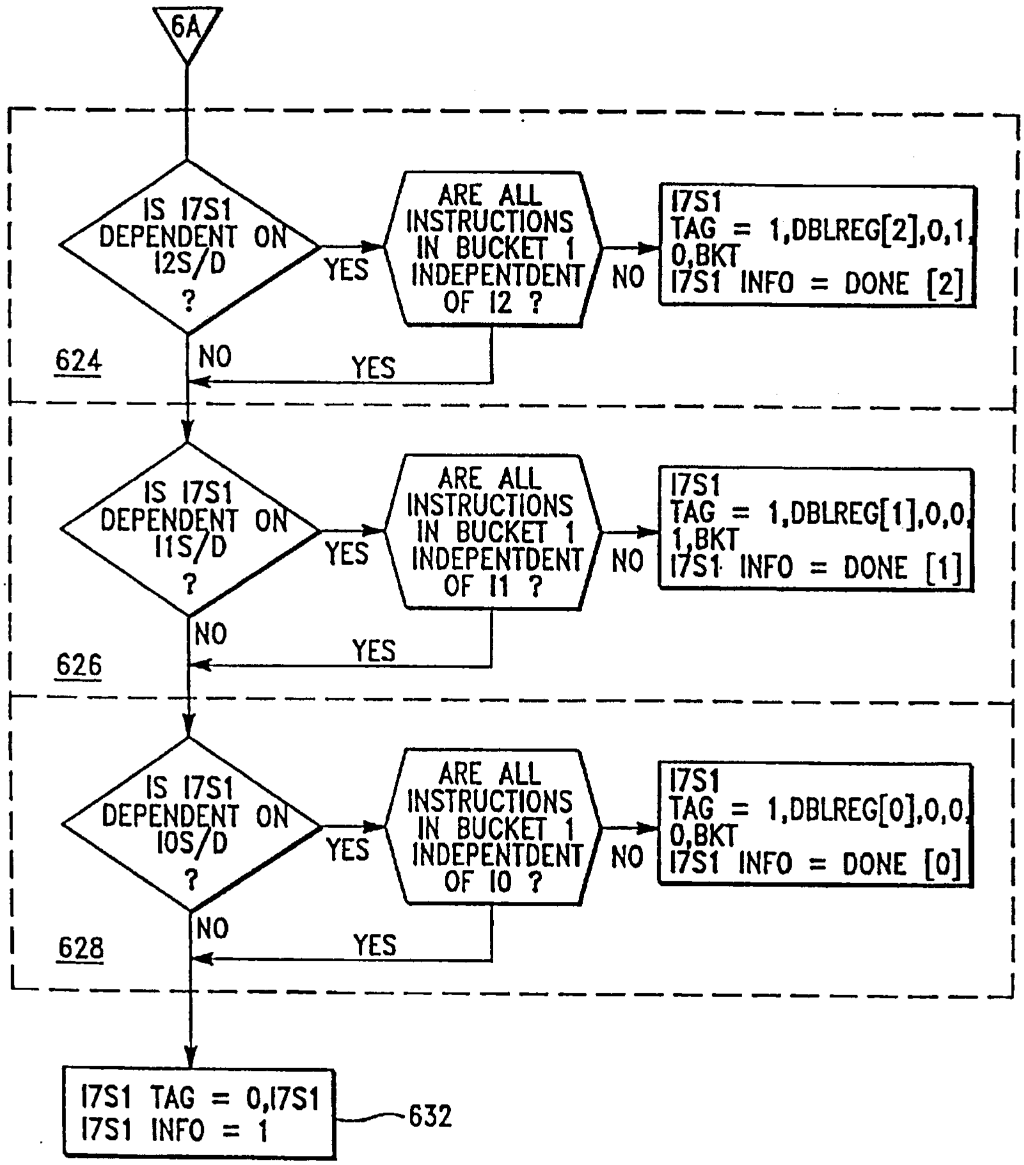

6A
IS I7S1 DEPENDENT ON I2S/D ?
YES
ARE ALL INSTRUCTIONS IN BUCKET 1 INDEPENTDENT OF I2 ?
NO
I7S1 TAG = 1,DBLREG[2],0,1, 0,BKT I7S1 INFO = DONE [2]
624
NO
YES
IS I7S1 DEPENDENT ON I1S/D ?
YES
ARE ALL INSTRUCTIONS IN BUCKET 1 INDEPENTDENT OF I1 ?
NO
I7S1 TAG = 1,DBLREG[1],0,0, 1,BKT I7S1 INFO = DONE [1]
626
NO
YES
IS I7S1 DEPENDENT ON I0S/D ?
YES
ARE ALL INSTRUCTIONS IN BUCKET 1 INDEPENTDENT OF I0 ?
NO
I7S1 TAG = 1,DBLREG[0],0,0, 0,BKT I7S1 INFO = DONE [0]
628
NO
YES
I7S1 TAG = 0,I7S1 I7S1 INFO = 1
632

FIG.—6B

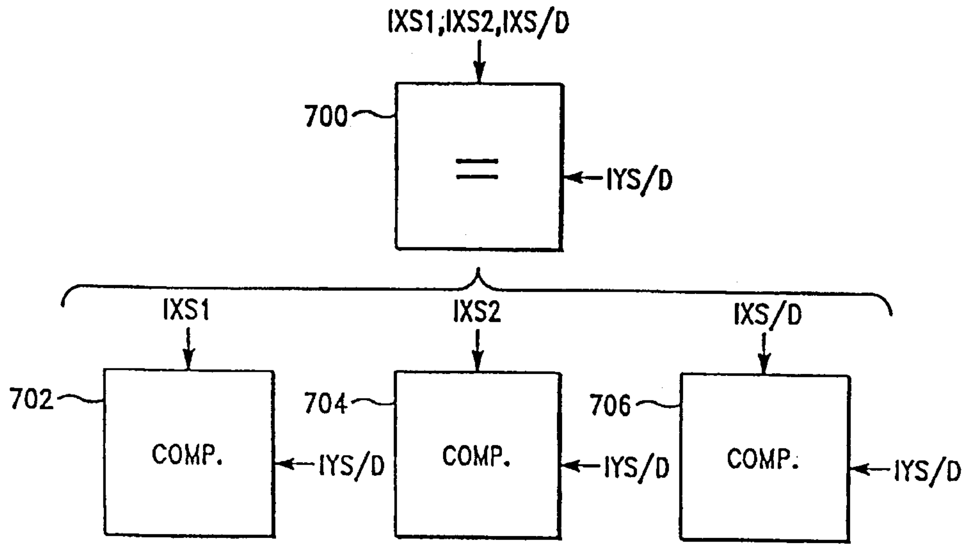

IXS1,IXS2,IXS/D
700
=
IYS/D
IXS1
IXS2
IXS/D
702
COMP.
IYS/D
704
COMP.
IYS/D
706
COMP.
IYS/D

FIG.—7

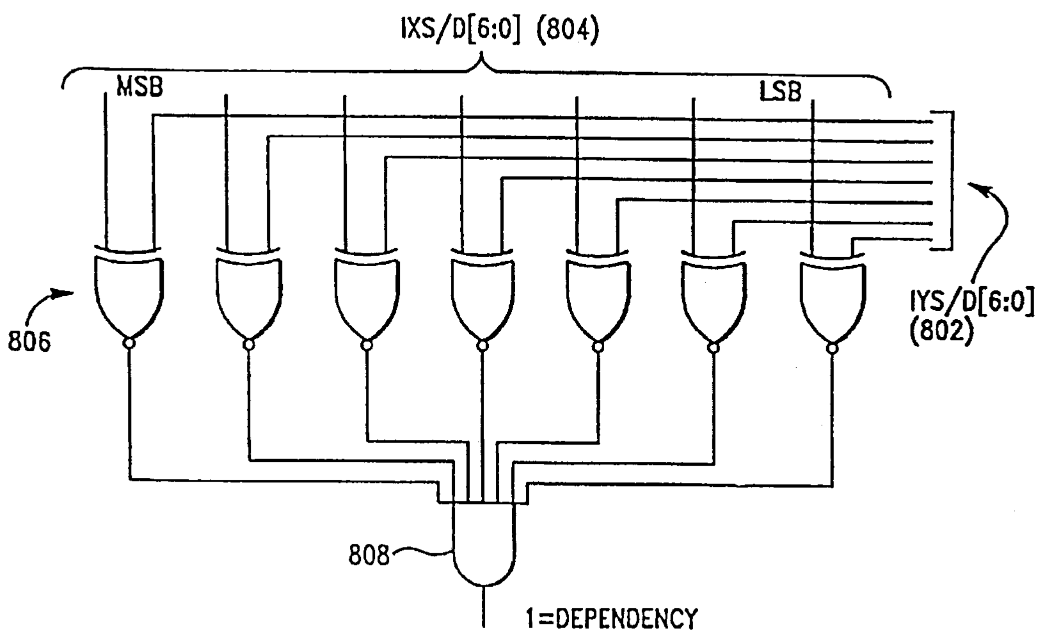

IXS/D[6:0] (804)
MSB
LSB
806
IYS/D[6:0]
(802)
808
1=DEPENDENCY

FIG.—8

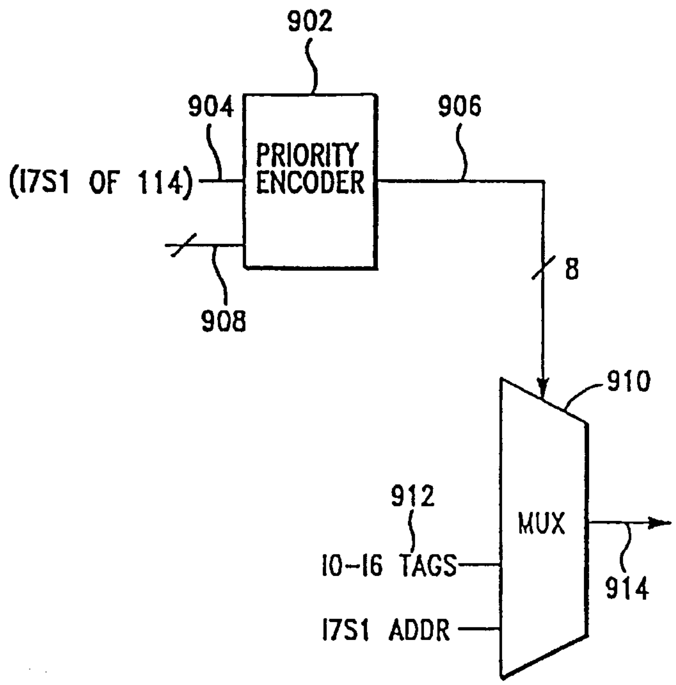

902
904
906
PRIORITY ENCODER
(I7S1 OF 114)
908
8
910
912
MUX
10-16 TAGS
I7S1 ADDR
914

FIG.—9

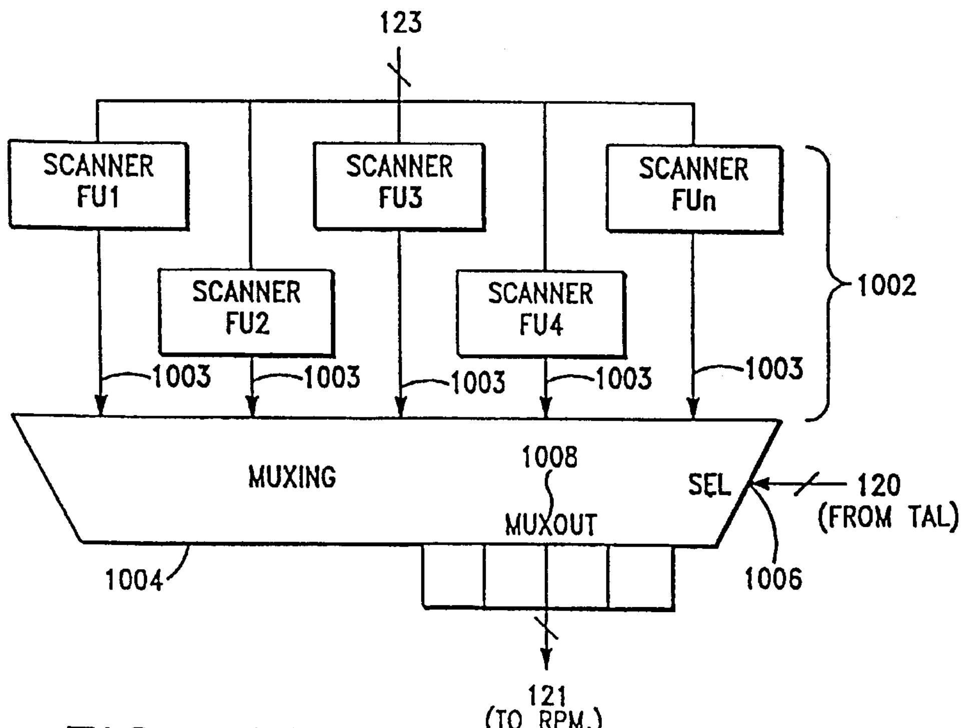

123
SCANNER FU1
SCANNER FU2
SCANNER FU3
SCANNER FU4
SCANNER FUn
1002
1003
MUXING
1008
SEL
120
(FROM TAL)
MUXOUT
1004
1006
121
(TO RPM.)

FIG.—10

SUPERSCALAR RISC INSTRUCTION SCHEDULING

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is a continuation of application Ser. No. 08/594,401, filed Jan. 31, 1996 (status; pending), which is a continuation of application Ser. No. 08/219,425, filed Mar. 29, 1994, now U.S. Pat. No. 5,497,499 which is a continuation of application Ser. No. 07/860,719, filed Mar. 31, 1992 (status: abandoned).

The following are commonly owned, applications: "Semiconductor Floor Plan and Method for a Register Renaming Circuit" Ser. No. 07/860,718, now U.S. Pat. No. 5,371,684 concurrently filed with the present application "High Performance RISC Microprocessor Architecture", Ser. No. 07/817,810, filed Jan. 8, 1992, now U.S. Pat. No. 5,539,911; "Extensible RISC Microprocessor Architecture", Ser. No. 07/817,809, filed Jan. 8, 1992, now abandoned. The disclosures of the above applications are incorporated herein by reference.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to superscalar reduced instruction set computers (RISC), more particularly, the present invention relates to instruction scheduling including register renaming and instruction issuing for superscalar RISC computers.

2. Related Art

A more detailed description of some of the basic concepts discussed in this application is found in a number of references, including Mike Johnson, Superscalar Microprocessor Design (Prentice-Hall, Inc., Englewood Ciffs, N.J., 1991); John L Hennessy et.al., Computer Architecure—A Quantitative Approach (Morgan Kaufmann Publishers, Inc., San Mateo, Calif., 1990). Johnson's text, particularly Chapters 2, 6 and 7 provide an excellent discussion of the register renaming issues addressed by the present invention.

A major consideration in a superscalar RISC processor is to how to execute multiple instructions in parallel and out-of-order, without incurring data errors due to dependencies inherent in such execution. Data dependency checking, register renaming and instruction scheduling are integral aspects of the solution.

2.1 Storage Conflicts and Register Renaming

True dependencies (sometimes called "flow dependencies" or "write-read" dependencies) are often grouped with anti-dependencies (also called "read-write" dependencies) and output dependencies (also called "write-write" dependencies) into a single group of instruction dependencies. The reason for this grouping is that each of these dependencies manifests itself through use of registers or other storage locations. However, it is important to distinguish true dependencies from the other two. True dependencies represent the flow of data and information through a program. Anti- and output dependencies arise because, at different points in time, registers or other storage locations hold different values for different computations.

When instructions are issued in order and complete in order, there is a one-to-one correspondence between registers and values. At any given point in execution, a register identifier precisely identifies the value contained in the corresponding register. When instructions are issued out of order and complete out of order, correspondence between register and values breaks down, and values conflict for register. The problem is severe when the goal of register allocation is to keep as many values in as few registers as possible. Keeping a large number of values in a small number of registers creates a large number of conflicts when the execution order is changed from the order assumed by the register allocator.

Anti- and output dependencies are more properly called "storage conflicts" because reusing storage locations (including registers) causes instructions to interfere with one another even though conflicting instructions are otherwise independent. Storage conflicts constrain instruction issue and reduce performance. But storage conflicts, like other resource conflicts, can be reduced or eliminated by duplicating the troublesome resource.

2.2 Dependency Mechanisms

Johnson also discusses in detail various dependency mechanisms, including: software, register renaming, register renaming with a reorder buffer, register renaming with a future buffer, interlocks, the copying of operands in the instruction window to avoid dependencies, and partial renaming.

A conventional hardware implementation relies on software to enforce dependencies between instructions. A compiler or other code generator can arrange the order of instructions so that the hardware cannot possibly see an instruction until it is free of true dependencies and storage conflicts. Unfortunately, this approach runs into several problems. Software does not always know the latency of processor operations, and thus, cannot always know how to arrange instructions to avoid dependencies. There is the question of how the software prevents the hardware from seeing an instruction until it is free of dependencies. In a scalar processor with low operation latencies, software can insert "no-ops" in the code to satisfy data dependencies without too much overhead. If the processor is attempting to fetch several instructions per cycle, or if some operations take several cycles to complete, the number of no-ops required to prevent the processor from seeing dependent instructions rapidly becomes excessive, causing an unacceptable increase in code size. The no-ops use a precious resource, the instruction cache, to encode dependencies between instructions.

When a processor permits out-of-order issue, it is not at all clear what mechanism software should use to enforce dependencies. Software has little control over the behavior of the processor, so it is hard to see how software prevents the processor from decoding dependent instructions The second consideration is that no existing binary code for any scalar processor enforces the dependencies in a superscalar processor, because the mode of execution is very different in the superscalar processor. Relying on software to enforce dependencies requires that the code be regenerated for the superscalar processor. Finally, the dependencies in the code are directly determined by the latencies in the hardware, so that the best code for each version of a superscalar processor depends on the implementation of that version.

On the other hand, there is some motivation against hardware dependency techniques, because they are inherently complex. Assuming instructions with two input operands and one output value, as holds for typical RISC instructions, then there are five possible dependencies between any two instructions: two true dependencies, two anti-dependencies, and one output dependency. Furthermore, the number of dependencies between a group of instructions, such as a group of instructions in a window,

varies with the square of the number of instructions in the group, because each instruction must be considered against every other instruction.

Complexity is further multiplied by the number of instructions that the processor attempts to decode, issue, and complete in a single cycle. These actions introduce dependencies. The only aid in reducing complexity is that the dependencies can be determined increment, over many cycles to help reduce the scope and complexity of the dependency hardware.

One technique for removing storage conflicts is by providing additional register that are used to reestablish the correspondence between registers and value. The additional registers are conventional allocated dynamically by hardware and the registers are associated with values needed by the program using "register renaming." To implement register renaming, processors typically allocate a new register for every new value produced (i.e., for every instruction that writes a register). An instruction identifying the original register, for the purpose of reading its value, obtains instead the value in the newly allocated register. Thus, hardware renames the original register identifier in the instruction to identify the new register and correct value. The same register identifier in several different instructions may access different hardware registers, depending on the locations of register references with respect to register assignments.

Consider the following code sequence where "op" is an operation, "Rn" represents a numbered register, and ":=" represents assignment

R3b :=R3a op R5a (1)

R4b :=R3b +1 (2)

R3c :=R5a +1 (3)

R7b :=R3c op R4b (4)

Each assignment to a register creates a new "instance" of the register, denoted by an alphabetic subscript. The creation of a new instance for R3 in the third instruction avoids the anti- and output dependencies on the second and first instructions, respectively, and yet does not interfere with correctly supplying an operand to the fourth instruction. The assignment to R3 in the third instruction supersedes the assignment to R3 in the first instruction, causing R3c to become the new R3 seen by subsequent instructions until another instruction assigns a value to R3.

Hardware that performs renaming creates each new register instance and destroys the instance when its value is superseded and there are no outstanding references to the value. This removes anti- and output dependencies and allows more instruction parallelism. Registers are still reused, but reuse is in line with the requirements of parallel execution. This is particularly helpful with out-of-order issue, because storage conflicts introduce instruction issue constants that are not really necessary to produce correct results. For example, in the preceding instruction sequence, renaming allows the third instruction to be issued immediately, whereas, without renaming, the instruction must be delayed until the first instruction is complete and the second instruction is issued.

Another technique for reducing dependencies is to associate a single bit (called a "scoreboard bit") with each register. The scoreboard bit is used to indicate that a register has a pending update. When an instruction is decoded that will write a register, the processor sets the associated scoreboard bit. The scoreboard bit is reset when the write actually occurs Because there is only one scoreboard bit indicating whether or not there is a pending update, there can be only one such update for each register. The scoreboard stalls instruction decoding if a decoded instruction will update a register that already has a pending update (indicated by the scoreboard bit being set). This avoids output dependencies by allowing only one pending update to a register at any given time.

Register renaming, in contrast, uses multiple-bit tags to identify the various uncomputed values, some of which values may be destined for the same processor register (that is, the same program-visible register). Conventional renaming requires hardware to allocate tags from a pool of available tags that are not currently associated with any value and requires hardware to free the tags to the pool once the values have been computed. Furthermore, since scoreboarding allows only one pending update to a given register, the processor is not concerned about which update is the most recent.

A further technique for reducing dependencies is using register renaming with a reorder buffer which uses associative lookup. The associative lookup maps the register identifier to the reorder buffer entry as soon as the entry is allocated, and, to avoid output dependencies the lookup is prioritized so that only the value for the most recent assignment is obtained if the register is assigned more than once. A tag is obtained if the result is not yet available. There can be as many instances of a given register as there are reorder buffer entries, so there are no storage conflicts between instructions. The values for the different instances are written from the reorder buffer to the register file in sequential order. When the value for the final instance is written to the register file, the reorder buffer no longer maps the register, the register file contains the only instance of the register, and this is the most recent instance.

However, renaming with a reorder buffer relies on the associative lookup in the reorder buffer to map register identifiers to values. In the reorder buffer, the associative lookup is prioritized so that the reorder buffer always provides the most recent value in the register of interest (or a tag). The reorder buffer also writes values to the register file in order, so that, if the value is not in the reorder buffer, the register file must contain the most recent value.

In a still further technique for reducing dependencies, associative lookup can be eliminated using a "future file." The future file does not have the properties of the reorder buffer discussed in the preceding paragraph. A value presented to the future file to be written may not be the most recent value destined for the corresponding register, and the value cannot be treated as the most recent value unless it actually is. The future file therefore keeps track of the most recent update and checks that each write corresponds to the most recent update before it actually performs the write.

When an instruction is decoded, it accesses tags in the future file along with the operand values. If the register has one or more pending updates, the tag identifies the update value required by the decoded instruction. Once an instruction is decoded, other instructions may overwrite this instructions's source operands without being constrained by anti-dependencies, because the operands are copied into the instruction window. Output dependencies are handled by preventing the writing as a result into the future file if the result does not have a tag for the most recent value. Both anti- and output dependencies are handled without stalling instruction issue.

If dependencies are not removed through renaming, "interlocks" must use to enforce dependencies. An interlock simply delays the execution of an instruction until the instruction is free of dependencies. There are two ways to prevent an instruction from being executed: one way is to

prevent the instruction from being decoded, and the other is to prevent the instruction from being issued.

To improve performance over scoreboarding, interlocks are moved from the decoder to the instruction window using a "dispatch stack." The dispatch stack is an instruction window that augments each instruction in the window with dependency counts. There is a dependency count associated with the source register of each instruction in the window, giving the number of pending prior updates to the source register and thus the number of updates that must be completed before all possible true dependencies are removed. There are two similar dependency counts associated with the destination register of each instruction in the window, giving both the number of pending prior uses of the register (which is the number of anti-dependencies) and the number of pending prior updates to the register (which is the number of output dependencies).

When an instruction is decoded and loaded into the dispatch stack, the dependency counts are set by comparing the instruction's register identifiers with the register identifiers of all instructions already in the dispatch stack. As instructions complete, the dependency counts of instructions that are still in the window are decremented based on the source and destination register identifiers of completing instructions (the counts are decremented by a variable amount, depending on the number of instructions completed). An instruction is independent when all of its counts are zero. The use of counts avoids having to compare all instructions in the dispatch stack to all other instructions on every cycle.

Anti-dependencies can be avoided altogether by copying operands to the iron window (for example, to the reservation stations) during instruction decode. In this manner, the operands cannot be overwritten by subsequent register updates. Operands can be copied to eliminate anti-dependencies in any approach, independent of register renaming. The alternative to coping operands is to interlock anti-dependencies, but the comparators and/or counters required for these interlocks are costly, considering the number of combinations of source and result registers to be compared.

A tag can be supplied for the operand rather than the operand itself. This tag is simply a means for the hardware to identify which value the instruction requires, so that, when the operand value is produced, it can be matched to the instruction. If there can be only one pending update to a register, the register identifier can serve as a tag (as with scoreboarding). If there can be more than one pending update to a register (as with renaming), there must be a mechanism for allocating result tags and insuring uniqueness.

An alternative to scoreboarding interlocking is to allow multiple pending updates of registers to avoid stalling the decoder for output dependencies, but to handle anti-dependencies by copying operands (or tags) during decode. An instruction in the window is not issued until it is free of output dependencies, so the updates to each register are performed in the same order in which they would be performed with in-order completion, except that updates for different registers are out of order with respect to each other. The alternative has almost all of the capabilities of register renaming, lacking only the capability to issue instructions so that updates to the same register occur out of order.

There appears to be no better alternative to renaming other than with a reorder buffer. Underlying the discussion of dependencies has been the assumption that the processor performs out-of-order issue and already has a reorder buffer for recovering from mispredicted branches. Out-of-order issue makes it unacceptable to stall the decoder for dependencies If the processor has an instruction window, it is inconsistent to limit the look ahead capability of the processor by interlocking the decoder. There are then only two alternatives: implement anti- and output dependency interlock in the window or remove these altogether with renaming.

SUMMARY OF THE INVENTION

The present invention is directed to instruction scheduling including register renaming and instruction issuing for superscalar RISC computers. A Register Rename Circuit (RRC), which is part of the scheduling logic allows a computer's Instruction Execution Unit (IEU) to execute several instructions at the same time while avoiding dependencies In contrast to conventional register renaming the present invention does not actually rename register addresses. The RRC of the present invention temporarily buffers the instruction results, and the results of out-of-order instruction execution are not transferred to the register file until all previous instructions are done. The RRC also performs result forwarding to provide temporarily buffered operands (results) to dependant instructions. The RRC contains three subsections: a Data Dependency Checker (DDC), Tag Assign Logic (TAL) and Register file Port MUXes (RPM).

The function of the DDC is to locate the dependencies between the instructions for a group of instructions. The DDC does this by comparing the addresses of the source registers of each instruction to the addresses of the destination registers of each previous instruction in the group. For example, if instruction A reads a value from a register that is written to by instruction B, then instruction A is dependent upon instruction B and instruction A cannot start until instruction B has finished. The DDC outputs indicate these dependencies.

The outputs of the DDC go to the TAL. Because it is possible for an instruction to be dependent on more than one previous instruction, the TAL must determine which of those previous instructions will be the last one to be executed. The present invention automatically maps each instruction a predetermined temporary buffer location hence, the present invention does not need prioritized associative look-up as used by convention reorder buffers, thereby saving chip area/cost and execution speed.

Out-of-order results for several instructions being executed at the same time are stored in a set of temporary buffers, rather that the file register designated by the instruction. If the DDC determines, for example, that a register that instruction **6**'s source is written to by instructions **2**, **3** and **5**, then the TAL will indicate that instruction **6** must wait for instruction **5** by outputting the "tag" of instruction **5** for instruction **6**. The tag of instruction **5** shows the temporary buffer location where instruction **5**'s result is stored. It also contains a one bit signal (called a "done flag") that indicates if instruction S is finished or not. The TAL will output three tags for each instruction, because each instruction can have three source registers. If an instruction is not dependent on any previous instruction, the TAL will output the register file address of the instruction's input, rather an a temporary buffer's address.

The last part of the RRC are the RPMs or Register file Port MUXes. The inputs of the RPMs are the outputs of the TAL, and the select lines for the RPMs come from another part of the IEU called the Instruction Scheduler or Issuer. The

Instruction Scheduler chooses which instruction to execute (this decision is based partly on the done flags) and then uses the RPMs to select the tags of that instruction. These tags go to the read address ports of the computer's register files. In the previous example, once instruction S has finished, the Instruction Scheduler will start instruction **6**. It will select the RPM so that the address of instruction **5**'s result (its tag) is sent to the register file, and the register file will make the result of instruction S available to instruction **6**.

The foregoing and other features and advantages of the present invention will be apparent from the following more particular description of the preferred embodiments of the invention, as illustrated in the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention will be better understood if reference is made to the accompanying drawings.

FIG. **1** shows a representative high level block diagram of the register renaming circuit of the present invention.

FIG. **2** shows a representative block diagram of the data dependency check circuit of the present invention.

FIG. **3** shows a representative block diagram of the tag assignment logic of the present invention.

FIG. **4** shows a representative block diagram of the register port file multiplexers of the present invention.

FIG. **5** is a representative flowchart showing a data dependency check method for IXS1 and IYS/D in accordance with the present invention.

FIGS. **6**A and **6**B are representative flowcharts showing a tag assignment me in accordance with the present invention.

FIG. **7** shows a representative block diagram which compares an instruction Y's source/destination operand with each operand of an instruction X accordance with an embodiment of the present invention.

FIG. **8** shows a representative circuit diagram for comparator block **706** of FIG. **7**.

FIG. **9** shows a representative block diagram of a Priority Encoder in accordance with an embodiment of the present invention.

FIG. **10** shows a representative block diagram of the instruction scheduling logic of the present invention.

DETAILED DESCRIPTION

FIG. **1** shows a representative high level block diagram of an Instruction Execution Unit (IEU) **100** associated with the present invention. The goal of IEU **100** is to execute as many instructions as possible in the shortest amount of time. There are two basic ways to accomplish this optimize IEU **100** so that each instruction takes as little time as possible or optimize IEU **100** so that it can execute several instructions at the same time.

Instructions are sent to IEU **100** from an Instruction Fetch Unit (IFU, not shown) through an instruction FIFO (first-in-first-out register stack storage device) **101** in groups of four called "buckets." IEU **100** can decode and schedule up to two buckets of instructions at one time. FIFO **101** stores **16** total instructions in four buckets labeled **0–3**. IEU **100** looks at the an instruction window **102**. In one embodiment of the present invention, window **102** comprises eight instructions (buckets **0** and **1**). Every cycle IEU **100** tries to issue a maximum number of instructions from window **102**. Window **102** functions as a instruction buffer register. Once the instructions in a bucket are executed and their results stored in the processor's register file (see block **117**), the bucket is flushed out a bottom **104** and a new bucket is dropped in at a top **106**.

In order to execute instructions in parallel or out of order, care must be taken so that the data that each instruction needs is available when the instruction needs it and also so that the result of each instruction is available for any future instructions that might need it. A Register Rename Circuit (RRC), which is part of the scheduling logic of the computer's IEU performs this function by locating dependencies between current instructions and then renaming the sources (inputs) of the instruction.

As noted above, there are three types of dependencies: input dependenies output dependencies and anti-dependencies. Input dependencies occur when an instruction, call it A, that performs an operation on the result of a previous instruction, call it B. Output dependencies occur when the outputs of A and B are to be stored in the same place. Anti-dependencies occur when instruction A comes before B in the instruction stream and B's result will be stored in the same place as one of A's inputs.

Input dependencies are handled by not executing instructions until their inputs are available. RRC **112** is used to locate the input dependencies between current instructions and then to signal an Instruction Scheduler or Issuer **118** when all inputs for a particular instruction are ready. In order to locate these dependencies, RRC **112** compares the register file addresses of each instruction's inputs with the addresses of each previous instructions output using a data dependency circuit (DDC) **108**. If one instruction's input comes from a register where a previous instruction's output will be stored, then the latter instruction must wait for the former to finish.

This implementation of RRC **112** can check eight instructions at the same time, so a current instruction is defined as any one of those eight from window **102**. It should become evident to those skilled in the art that the present invention can easily be adapted to check more or less instructions.

In one embodiment of the present invention, instructions can have from **0** to **3** inputs and **0** or **1** outputs. Most instructions' inputs and outputs come from, or are stored in, one of several register files. Each register file **117** (e.g., separate integer, floating and boolean register files) has 32 real entries plus the group of **8** temporary buffers **116**. When an instruction completes, (The term "complete" means that the operation is complete and the operand is ready to be written to its destination register.) its result is stored in its preassigned location in the temporary buffers **116**. Its result is later moved to the appropriate place in register file **117** after all previous instructions' results have been moved to their places in the register file. This movement of results from temporary buffers **116** to register file **117** is called "retirement" and is controlled by termination logic, as should become evident to those skilled in the art. More than one instruction may be retired at a time. Retirement comprises updating the "official state" of the machine, including the computer's Program. Counter, as will become evident to those skilled in the art. For example, if instruction **I0** happens to complete directly before instruction I1, both results can be stored directly into register file **117**. But if instruction I3 then completes, its result must be stored in temporary buffer **116** until instruction I2 completes. By having IEU **100** store each instruction's result in its preassigned place in the temporary buffers **116**, IEU **100** can execute instructions out of program order and still avoid the problems caused by output and anti-dependencies.

RRC **112** sends a bit map to an Instruction Scheduler **118** via a bus **120** indicating which instructions in window **102**

are ready for issuing. Instruction decode logic (not shown) indicates to Issuer **118** the resource requirements for each instruction over a bus **123**. For each resource in IEU **100** (e.g., each functional unit being an adder, multiplier, shifter, or the like), Issuer **118** scans this information and selects the first and subsequent instructions for issuing by sending issue signals over bus **121**. The issue signals select a group of Register File Port MUXes (RPMs) **124** inside RRC **112** whose inputs are the addresses of each instruction's inputs.

Because the results may stay in temporary buffer **116** several cycles before going to register file **117**, a mechanism is provided to get results from temporary buffer **116** before they go to register file **117**, so the information can be used as operands for other instructions. This mechanism is called "result forwarding," and without it, Issuer **118** would not be able to issue instructions out of order. This result forwarding is done in register file **117** and is controlled by RRC **112**. The control signals necessary for performing the result forwarding will be come evident to those skilled in the art, as should the random logic used for generating such control signals.

If an instruction is not dependent on any of the current instructions result forwarding is not necessary since the instruction's inputs are already in register file **117**. When Issuer **118** decides to execute that instruction, RRC **112** tells register file **117** to output its data.

RRC **112** contains three subsections: a Data Dependency Checker (DDC) **108**, Tag Assign Logic (TAL) **122** and Register File Port MUXes (RPM) **124**. DDC **108** determines where the input dependencies are between the current instructions. TAL **122** monitors the dependencies for Issuer **118** and controls result forwarding. RPM **124** is controlled by Issuer **118** and directs the outputs of TAL **122** to the appropriate register file address ports **119**. Instructions are passed to DDC **108** via bus **110**. All source registers are compared with all previous destination registers for each instruction in window **102**.

Each instruction has only one destination, which may be a double register in one embodiment An instruction can only depend on a previous instruction and may have up to three source registers. There are various register file source and destination addresses that need to be checked against each other for any dependencies. As noted above, the eight bottom instructions corresponding to the lower two buckets are checked by DDC **108**. All source register addresses are compared with all previous destination register addresses for the instructions in window **102**.

For example, let's say a program has the following instruction sequence:

add R**0**, R**1**, R2 (0)

add R**0**, R2, R3 (1)

add R4, R**5**, R2 (2)

add RZ, R3, R4 (3)

The first two registers in each instruction **0–3** are the source registers, and the last listed register in each instruction is the destination register. For example, R**0** and R1 arc the source registers for instruction **0** and R2 is the destination register. Instruction **0** adds the contents of registers **0** and 1 and stores the result in R2. For instructions 1–3 in this dependencies:

I1S1, I1S2 vs. I0D

I2S1, I2S2 vs. I1D, I0D

I3S1, I3S2 vs. I2D, I1D, I0D

The key to the above is as follows: IXRS1 is the address of source (input) number 1 of instruction X; IXRS2 is the address of source (input) number **2** of instruction X; and IXD is the address of the destination (output) of instruction X.

Note also that RRC **112 547** can ignore the fact that instruction **2** is output dependent on instruction **0**, because the processor has a temporary buffer where instruction **2**'s result can be stored without interfering with instruction **0**'s result. As discussed before, instruction **2**'s result will not be moved from temporary buffers **116** to register file **117** until instructions **0** and **1**'s results are moved to register file **117**.

The number of instructions that can be checked by RRC **112** is easily scaleable. In order to check eight instructions at a time instead of four, the following additional comparisons would also need to be made:

I4S1, I4S2 vs I3D, I2D, I1D, I9D

IS51, I5S2 vs 14D, I3D, I2D, I1D, I0D

I6S1, I6S2 vs I5D, I4D, I3D, 12D, I1D, I0D

I7S1, I7S2 vs I6D, I5D, I4D, I3D, I2D, I1D, T0D

There are several special cases that RRC **112** must handle in order to do the dependency check. First, there are some instructions that use the same register as an input and an output. Thus, RRC **112** must compare this source/destination register address with the destination register addresses of all previous instructions. So for instruction **7**, the following comparisons would be necessary:

I7S1,I7S2,I7S/D vs. I6D,I5D,I4D,I3D,I2D,I1I,I0D.

Another special case occurs when a program contains instructions that generate 64 bit outputs (called long-word operations). These instructions need two registers in which to store their results. In this embodiment, these registers must be sequential. Thus if RRC **112** is checking instruction **4**'s dependencies and instruction **1** is a long-word operation, then it must do the following comparisons:

I4S1,I4S2 vs. I3D,I2D,I1D,ID+1,I0D

Sometimes, instructions do not have destination registers. Thus RRC **112** must ignore any dependencies between instructions without destination registers and any future instructions. Also, instructions may not have only one valid source register, so RRC **112** must ignore any dependencies between the unused source register (usually S2) and any previous instructions.

RRC **112** is also capable of dealing with multiple register files. When using multiple register files, dependencies only occur when one instruction's source register has the same address and is in the same register file as some other instruction's destination register. RRC **112** treats the information regarding which register file a particular address is from as part of the address. For example, in an implementation using four 32 bit register files, RRC **112** would do 7 bit compares instead of 5 bit compares (5 for the address and 2 for the register file).

Signals indicating which instructions are long-word operations or have invalid source or destination registers are sent to RRC **112** from Instruction Decode Logic (IDL; not shown). IDL also tells RRC **112** which register file each instruction's sources and destinations will come from or go to.

A block diagram of DDC **108** is shown in FIG. **2**. Source address signals arrive from IFIFO **101** for all eight instructions of window **102**. Additional inputs include long-word load operation flags, register file decode signals, invalid destination register flags, destination address signals and addressing mode flags for all eight instructions.

DDC **208** comprises **28** data dependency blocks **204**. Each block **204** is described in a KEY **206**. Each block **204** receives 3 inputs, IXS1, IXSD and IXS/D. IXS1 is the address of source (input) number 1 of instruction X, IXS2 is the address of source (input) number **2** of instruction X and IXS/D is the address of the source/destination (input) of

instruction X Each block **204** also receives input IYS/D, which is the destination register address for some previous instruction Y. A top row **208**, for example, receives IOS/D, which is the destination register address for instruction **0**. Each block **204** outputs the data dependency results to one of a corresponding bus line **114**. For example, the address of I2S/D must be checked with operand addresses S1, S2 and S/D of instructions **7**, **6**, **5**, **4**, and **3**.

Each block **204** performs the three comparisons. To illustrate these comparisons, consider a generic block **700** shown in FIG. **7**, which compares instruction Y's source/destination operand with each operand of instruction X In this example, the three following comparisons must be made:

IXS1=IYS/D

IXS2=IYS/D

IXS/D=IYS/D

These comparisons are represented by three comparator blocks **702**, **704** and **706**, respectively. One set of inputs to comparator blocks **702**, **704** and **706** are the bits of the IYS/D field, which is represented by number **708**. Comparator block **702** has as its second set of inputs the bits of the IXS1. Similarly, comparator block **704** has as its second set of inputs the bits of the IXS1, and comparator block **706** has as its second set of inputs the bits of the IXS/D.

In a preferred embodiment, the comparisons performed by blocks **702**, **704** and **706** can be performed by random logic. An example of random logic for comparator block **706** is shown in FIG. **8**. Instruction Y's source/destination bits [**6**:**0**] are shown input from the right at reference number **802** and instruction X's source/destination bits [**6**:**0**] are shown input from the top at reference number **804**. The most scant bit (MSB) is bit **6** and the least significant bit (LSB) is bit **0**. The corresponding bits from the two operands are fed to a set of seven exclusive NOR gates (XNORS) **806**. The outputs of XNORs **806** are then ANDed by a seven input AND gate **808**. If the corrresponding bits are the same, the output of XNOR **806** will be logic high. When all bits are the same, all seven XNOR **806** outputs are logic high and the output of AND gate **808** is logic high, this indicates that there is a dependency between IXS/D and IYS/D.

The random logic for comparator blocks **702** and **704** will be identical to that shown in FIG. **8**. The present invention contemplates many other random logic circuits for performing data dependency checking, as will become evident to those skilled in the art without departing from the spirit of this example.

As will further become evident to those skilled in the art, various implementation specific special cases can arise which require additional random logic to perform data dependency checking. An illustrative special data dependency checking case is for long word handling.

As mentioned before, if a long word operation writes to register X, the first 32 bits are written to register X and the second 32 bits are written to register X+1. The data dependency checker therefore needs to check both resisters when doing a comparison. In a preferred embodiment, register X is an even register, X+1 is an odd register and thus they only differ by the LSB. The easiest way to check both registers at the same time is to simply ignore the LSB. In the case of a store long (STLG) or load long (LDLG) operation, if X and Y only differ by the LSB bit [**0**], the logic in FIG. **8** would cause there to be no dependency, when there really is a dependency. Therefore, for a long word operation the STLG and LDLG flags must be ORed with the output of the [**0**] bit XNOR to assure that all dependencies are detected.

A data dependency check flowchart for IXS1 and IYS/D is shown in FIG. **5**. DDC **108** first checks whether IXS1 and IYS/D are in the same register file, as shown at a conditional block **502**. If they are not in the same register file there is no dependency. This is shown at block a **504**. If there is a dependency, DDC **108** then determines whether IXS1 and IYS/D are in the same register, as shown at a block **506**. If they are not in the same register, flow proceeds to a conditional block **508** where DDC **108** determines whether IY is a long word operation. If IY is not a long word operation there is no dependency and flow proceeds to a block **504**. If IY is a long word operation, flow then proceeds to a conditional statement **510** where DDC **108** determines whether IXS1 and IYS/D+1 are the same register. If they are not, there is no dependency and flow proceeds to a block **504**. If IXS1 and IYS/D+1 are the same register, flow proceeds to a conditional block **512** where DDC **108** determines if IY has a valid destination. If it does not have a valid destination, there is no dependency and flow proceeds to block **504**. If IY does have a valid destination, flow proceeds to a conditional block **514** where DDC **108** determines if IXS1 has a valid source register. Again, if no valid source register is detected there is no dependency, and flow proceeds to a block **504**. If a valid source register is detected, DDC **108** has determined that there is a dependency between IXS1 and IYX/D, as shown at a block **516**.

A more detailed discussion of data dependency checking is found in commonly owned, copending application Ser. No. 07/860,718 (Attorney Docket No. SP041/1397.0190000), the disclosure of which is incorporated herein by reference.

Because it is possible that an instruction might get one of its inputs from a register that was written to by several other instructions, the present invention must choose which one is the real dependency. For example, if instructions **2** and **5** write to register **4** and instruction **7** reads register **4**, then instruction **7** has two possible dependencies. In this case, it is assumed that since instruction **5** came after instruction **2** in the program, the programmer intended instruction **7** to use instruction **5**'s result and not instruction **2**'s. So, if an instruction can be dependent on several previous instructions, RRC **112** will consider it to be dependent on the highest numbered previous instruction.

Once TAL **122** has determined where the real dependencies are, it must locate the inputs for each instruction. In a preferred embodiment of the present invention, the inputs can come from the actual register file or an array temporary buffers **116**. RRC **112** assumes that if an instruction has no dependencies, its inputs are all in the register file. In this case, RRC **112** passes the IXS1, IXS2 and IXS/D addresses that came from IFIFO **102** to the register file. If an instruction has a dependency, then RRC **112** assumes that the data is in temporary buffers **116**. Since RRC **112** knows which previous instruction each instruction depends on, and since each instruction always writes to the same place in temporary buffers **116**, RRC **112** can determine where in temporary buffers **116** an instruction's inputs are stored. It sends these addresses to register file read ports **119** and register file **117** outputs the data from temporary buffers **116** so that the instruction can use it.

The following is an example of tag assignments:

0: add r**0**, r**2**, r**2**

1: add r**0**, r**2**, r**3**

2: add r**4**, r**5**, r**2**

3: add r**2**, r**3**, r**4**

The following are the dependencies for the above operations (dependencies are represented by the symbol "#"):

I1S2#IOS/D

I3S1#IOS/D

I3S1#I2S/D

I3S2#I1S/D

First, look at I0; since it has no dependencies, its tags are equal to its original source register addresses:

I0S1 TAG=I0S1=r0

I0S2 TAG=I0S2=r1

I0S/D TAG=I0S/D=r2

I1 has one dependency, and its tags are as follows:

I1S1 TAG=I1S1=t0

I1S2 TAG=I0S/D=t0 where: (t0=inst 0's slot in temporary buffer)

I1S/D TAG=I1S/D=r3

I2 is also independent

I2S1 TAG=I2S1=r4

I2S2 TAG=I2S2=r5

I2S/D TAG=I2S/D=r2

I3S1 has two possible dependencies, IOS/D and I2SD. Because TAL 122 must pick the last one (highest numbered one), I2S/D is chosen.

I3S1 TAG=I2S/D=t2

I3S2 TAG=I15/D=t1

I3S/D TAG=I3S/D=r4

These tags are then sent to RPM 124 via bus 126 to be selected by Issuer 118. At the same time TAL 122 is preparing the tags, it is also monitoring the outputs of DCL 130 and passing them on to Issuer 118 using bus 120. TAL 122 chooses the proper outputs of DCL's 130 to pass to Issuer 118 by the same method that it chooses the tags that it sends to RPM 124.

Continuing the example, TAL 122 sends the following ready signals to Issuer 118:

IOS1 INFO = 1
    (Inst 0 is independent so it can start immediately)
IOS2 INFO = 1
IOS/D INFO = 1
I1S1 INFO = 1
I1S2 INFO= DONE[0]
    (DONE[0] = 1 when IO is done)
I1S/D INFO = 1
I2S1 INFO = 1
I2S2 INFO = 1
I2S/D INFO = 1
I3S1 INFO = DONE[2]
I3S2 INFO = DONE[1]
I3S/D READ = 1

(The DONE signals come from DCL 130 via a bus 132. In connection with the present invention, the term "done" means the result of the instruction is in a temporary buffer or otherwise available at the output of a functional unit. Contastingly, the term "terminate" means the result of the instruction is in the register file.)

Turning now to FIG. 3, a representative block diagram of TAL 122 will be discussed. TAL 122 comprises 8 tag assignment logic blocks 302. Each TAL block 302 receives the corresponding data dependency results via buses 114, as well as further signals that come from the computer's Instruction Decode and control logic (not shown). The BKT bit signal forms the least significant bit of the tag. DONE[X] flags are for instructions 0 through 6, and indicate if instruction X is done. DBLREG[X] flags indicates which, if any, of the instructions is a double (long) word. Each TAL block 302 also receives its own instructions register addresses as

inputs. The Misc. signals, DBLREG and BKT signals are all implementation dependent control signals. Each TAL block 302 outputs 3 TAGs 126 labeled IXS1, IXM2 and IXS/D, which are 6 bits. TAL 122 outputs the least significant 5 bits of each TAG signal to RPMs 124 and the most significant TAG to Issuer 118.

Each block 302 of FIG. 3 comprises three Priority Encoders (PE), one for S1, one for S2 and one for S/D. There is one exception however. I0 requires no tag assignment. Its tags are the same as the original S1, S2 and S/D addresses, because I0 is always independent.

An illustrative PE is shown in FIG. 9. PE 902 has eight inputs 904 and eight outputs 906. Inputs 904 for PE 902 are outputs 114 from DDC 108 which show where dependencies exist. For example, in the case of source register 1 (S1), I7S1 tag assign PE 902's seven inputs are the seven outputs 114 of DDC 108 that indicate whether i7S1 is dependent on I6D, whether I7S1 is dependent on I5D, and so on down to whether I7S1 is dependent on I0D. An eighth input, shown at reference number 908, is always tied high because there should always be an output from PE 902.

As stated before, if an instruction depends on several previous instructions, PE 902 will select and output only the most previous instruction (in program order) on which there is a dependency. This is accomplished by connecting the signal showing if there is a dependency on the most previous instruction to the highest priority input of the PE 902 and the signal showing if there is a dependency on the second most previous instruction to the input of PE 902 with the second highest priority and so on for all previous instructions. The input of the PE 902 with the lowest priority is always tied high so that at least one of PE 902's outputs will be asserted.

Outputs 906 are used as select lines for a MUX 910. MUX 910 has eight inputs 912 to which the tags for each instruction are applied.

To illustrate this, assume that I7 depends on I6 and I5, then, since I6 has a higher priority than I5, the bit corresponding to I6 at outputs 906 of PE 902 will be high. At the corresponding input 912 of MUX 910 will be I6's tag for S1 (recall PE 902 is for I7S1). Because I7 is dependent on I6, the location of I6's result must be output from MUX 910 so that it can be used by I7. I6's tag will therefore be selected and output on an output line 914. I6's done flag, DONE[6] must also be output from MUX 910 so that Issuer 118 will know when I7's input is ready. This data is passed to Issuer 118 via bus 120. Since an instruction can have up to three sources, TAL 122 monitors up to three dependencies for each instruction and sends three vectors for each instruction (totalling 24 vectors) to Issuer 118. If an instruction is independent, TAL 122 signals to Issuer 118 that the instruction can begin immediately.

The MSB of the tag outputs which are sent to RPMs 124 is used to indicate if the address is a register file address or a temporary buffer address. If an instruction is independent, then the five LSB outputs indicate the source register address. For instructions that have dependencies: the second MSB indicates that the address is for a 64 bit valve the third through fifth MSB outputs specify the temporary buffer address; and the LSB output indicates which bucket is the current bucket, which is equal to the BKT signal in TAL 122.

Like DDC 108, TAL 122 has numerous implementation dependent, (i.e., special cases) that it handles. First, in an embodiment of the present invention, register number 0 of the register file is always equal to 0. Therefore, even if one instruction writes to register 0 and another reads from register 0, there will be no dependency between them. TAL 122 receives three signals from Instruction Decode Logic

(IDL; not shown) for each instruction to indicate if one of that instruction's sources is register **0**. If any of those is asserted, TAL **122** will ignore any dependencies for that particular input of that instruction.

Another special case occurs because under some circumstances, an instruction in bucket **0** will be guaranteed to not have any of the instructions in bucket **1** dependent on it A four bit signal called BKT1_NODEP_is sent to RRC **112** from the IEU control logic (not shown) and if BKT1_NODEP[X]=1 then RRC **112** knows to ignore any dependencies between instructions, **4**,**5**,**6** or **7** and instruction X

An example for TAG assignment of instruction **7**'s source **1** (**I7S1**) is shown in a flowchart in FIGS. **6A**–**6B**. TAL **122** first determines whether **I7S1** is register **0**, as shown at a conditional block **602**. If the first source operand for **I7** is register **0**, the TAG is set equal to zero, and the **I7S1**'s INFO flag is set equal to one, as shown in a block **604**. If the first source operand (S1) for **I7** is not register **0**, TAL **122** then determines if **I7S1** is dependent on **I6S/D**, as shown at a conditional block **606**. If **I7S1** is dependent on **I6S/D** flow then proceeds to a block **610** where **I7S1**'s TAG is set equal to {**1**,**DBLREG[6]**,**0**,**1**,**0**,**BKr**} and **I7S1**'s INFO flag is set equal to DONE[**6**], as shown at a block **610**. If either of the condition tested at a conditional block **606** is not met, flow proceeds to conditional block **612** where TAL **122** determines if **I7S1** is dependent on **I5S/D**. If there is a dependency, flow then proceeds to block **616** where TAL **122** sets **I7S1**'s TAG equal to {**1**,**DBLREG[5]**,**0**,**0**,**1**,**BKT**} and **I7S1**'s INFO flag is set equal to DONE[**5**]. If the condition tested at block **612** is not met, flow proceeds to a block **618** where TAL **122** determines if **I7S1** is dependent on **I4S/D**.

As evident by inspection of the remaining sections of FIGS. **6A** and **6B**, similar TAG determinations are made depending on whether **I7S1** is dependent on **I4S/D**, **I3S/D**, **I2SID**, **I1S/D** and **IOS/D**, as shown at sections **620**, **622**, **624**, **626** and **628**, respectively. Finally, if instruction **7** is independent of instruction **0** or if all instructions in bucket **1** are independent of instruction **0** (i.e., if BKI1_NODEP [**0**]=1), as tested at a conditional block **630**, the flow proceeds to block **632** where TAL **122** sets **I7S1**'s TAG equal to {**0**,**I7S1**} and **I7S1**'s INFO flag equal to 1. It should be noted for the above example that **I7S1** TAG signals are forwarded directly the register file port MUXes of register file **117**. The **I7S1** INFO signals are sent to Issuer **118** to tell it when **I7**'s S1 input is ready.

A representative block diagram of Issuer **118** is shown in FIG. **10**. In a preferred embodiment, Issuer **118** has one scanner block **1002** for each resource (functional unit) that has to be allocated. In this example, Issuer **118** has scanner blocks FU1, FU2, FU3, FU4 through FUn. Requests for functional units are generated from instruction information by decoding logic (not shown) in a known manner, which are sent to scanners **1002** via bus **123**. Each scanner block **1002** scans from instruction **I0** to **I7** and selects the first request for the corresponding functional unit to be serviced during that cycle.

In the case of multiple register files (integer, floating and/or boolean), Issuer **118** is capable of issuing instructions having operands stored in different register files. For example, an ADD instruction may have a first operand from the floating point register file and a second operand from the integer register file. Instructions with operands from different register files are typically given higher issue priority (i.e., they are issued first). This issuing technique conserves processor execution time and functional unit resources.

In a further embodiment in which IEU **100** may include two ALU's, ALU scanning becomes a bit more complicated. For speed reasons, one ALU scanner block scans from **I0** to **I7**, while the other scanner block scans from **I7** to **I0**. This is how two ALU requests are selected. With this scheme it is possible that an ALU instruction in bucket **1** will get issued before an ALU instruction in bucket **0**, while increasing scanning efficiency.

Scanner outputs **1003** are selected by MUXing logic **1004**. A set of SELect inputs **1006** for MUX **1004** receive three 8-bit vectors (one for each operand) from TAL **122** via bus **120**. The vectors indicate which of the eight instructions have no dependencies and are ready to be issued. Issuer **118** must wait for this information before it can start to issue any instructions. Issuer **118** monitors these vectors and when all three go high for a particular instruction, Issuer **118** knows that the inputs for that instruction are ready. Once the necessary functional unit is ready, the issuer can issue that instruction and send select signals to the register file port MUXes to pass the corresponding instructions outputs to register file **117**.

In a preferred embodiment of the present invention, after Issuer **118** is done it provides two 8-bit vectors per register file back to RRC **112** via MUXOUTputs **1008** to bus **121**. These vectors indicate which instructions are issued this cycle, are used a select lines for RPMs **124**.

The maximum number of instructions that can be issued simultaneously for each register file is restricted by the number of register file read ports available. A data dependency with a previous uncompleted instruction may prevent an instruction from being issued. In addition, an instruction may be prevented from being issued if the necessary functional unit is allocated to another instruction.

Several instructions, such as load immediate instructions, Boolean operations and relative conditional branches, may be issued. independently, because they may not require resources other than register file read ports or they may potentially have no dependencies.

The last section of RRC **112** is the register file port MUX (RPM) section **124**. The function of RPMs **124** is to provide a way for Issuer **118** to get data out of register files **117** for each instruction to use. RPMs **124** receive tag information via bus **126**, and the select lines for RPMs **124** come from Issuer **118** via a bus **121** and also from the computer's IEU control logic. The selected TAGs comprise read addresses that are sent to a predetermined set of ports **119** of register file **117** using bus **128**.

The number and design of RPMs **124** depend on the number of register files and the number of ports on each register file. One embodiment of RPMs **124** is shown in FIG. **4**. In this embodiment, RPMs **124** comprises **3** register port file MUXes **402**, **404** and **406**. MUX **402** receives as inputs the TAGs of instructions **0**–**7** corresponding to the source register field S1 that are generated by TAL **122**. MUX **404** receives as inputs the TAGs of instructions **0**–**7** corresponding to the source register field S2 that are generated by TAL **122**. MUX **406** receives as inputs the TAGs of instructions **0**–**7** corresponding to the source/destination register field S/D that are generated by TAL **122**. The outputs of MUXes **402**, **404** and **406** are connected to the read addresses ports of register file **117** via bus **128**.

RRC **112** and Issuer **118** allow the processor to execute instructions simultaneously and out of program order. An IEU for use with the present invention is disclosed in commonly owned, co-pending application Ser. No. 07/817,810 the disclosure of which is incorporated herein by reference.

While various embodiments of the present invention have been described above, it should be understood that they have

been presented by way of example, and not limitation. Thus the breadth and scope of the present invention should not be limited by any of the above-described exemplary embodiments, but should be defined only in accordance with the following claims and their equivalents.

What is claimed is:

1. In a computer system having a register file comprising a plurality of registers and a plurality of index-addressable temporary storage locations, a method for executing instructions having a prescribed program order, comprising the steps of:

(1) storing a plurality of instructions in an instruction buffer, wherein each instruction has an input and an output;

(2) assigning a unique one of the plurality of index-addressable temporary storage locations to each one of said plurality of instructions in said instruction buffer, wherein an output corresponding to a given one of said plurality of instructions is stored in said index-addressable temporary storage location assigned to said given one of said plurality of instructions;

(3) determining whether one of said plurality of instructions in said instruction buffer is a dependent instruction, wherein said dependent instruction has an input that is an output of a previous instruction, wherein said previous instruction is an instruction in said instruction buffer that precedes said dependent instruction in the prescribed program order; and

(4) associating said index-addressable temporary storage location assigned to said previous instruction with said input.

2. The method of claim 1, further comprising the steps of:

(5) executing said dependent instruction only after said previous instruction produces an output;

(6) storing said output in said index-addressable temporary storage location assigned to said previous instruction; and

(7) performing an operation corresponding to said dependent instruction using said output stored in said index-addressable temporary storage location assigned to said previous instruction as said input.

3. The method of claim 1, further comprising the step of associating a done signal with said input, wherein said done signal indicates a status of said previous instruction.

4. The method of claim 1, further comprising the step of storing said output in an appropriate register when said previous instruction is retired.

5. A superscalar processor for executing instructions having a prescribed program order, comprising:

an instruction buffer for storing a plurality of instructions;

an index-addressable temporary buffer comprising a plurality of temporary storage locations, wherein each one of said plurality of instructions is assigned to a unique one of said plurality of temporary storage locations, wherein an output corresponding to a given one of said plurality of instructions is stored in said temporary storage location assigned to said given one of said plurality of instructions;

a data dependency checker to locate a dependent instruction stored in said instruction buffer, wherein said dependent instruction has an input that is dependent on a previous instruction, wherein said previous instruction is an instruction in said instruction buffer that precedes said dependent instruction in the prescribed program order; and

a circuit that receives from said data dependency checker dependency data corresponding to said dependent instruction and uses said dependency data to associate said temporary storage location assigned to said previous instruction with said input.

6. The superscalar processor of claim 5, wherein said circuit associates said temporary storage location assigned to said previous instruction with said input by outputting a reference corresponding to said temporary storage location assigned to said previous instruction.

7. The superscalar processor of claim 6, wherein said reference comprises an address.

8. The superscalar process of claim 7, wherein said reference further comprises a 1-bit identifier that indicates whether said address is an address of one of said temporary storage locations or is an address of a register.

9. The superscalar processor of claim 7, wherein said circuit further outputs a first signal indicating a completion status of said previous instruction.

10. The superscalar processor of claim 9, further comprising:

a multiplexer having a plurality of inputs, wherein said multiplexer receives an address of a storage location at each one of said plurality of inputs, wherein one of said addresses received at one of said plurality of inputs is said address included in said reference outputted by said circuit; and

an issuer that receives as an input said signal indicating a completion status of said previous instruction and that outputs a select signal to said multiplexer.

11. The superscalar processor of claim 5, wherein said circuit comprises a priority encoder and a multiplexer, wherein said encoder receives said dependency data, and an output of said encoder is used as a select signal for said multiplexer.

12. The superscalar processor of claim 11, wherein said multiplexer receives a plurality of references as inputs and outputs a reference corresponding to said output of said encoder, and wherein said reference outputted by said multiplexer represents an address of said temporary storage location assigned to said previous instruction.

13. The superscalar processor of claim 5, wherein said instruction buffer is capable of storing at most X number of instructions, and said temporary buffer includes at least X number of temporary storage locations, wherein X is a positive integer.

14. The superscalar processor of claim 5, wherein one of said plurality of instructions is assigned to a unique one of said plurality of storage locations based on a position of said one of said plurality of instructions within said instruction buffer.

15. The superscalar processor of claim 5, wherein said data dependency checker locates a dependent instruction stored in said instruction buffer by comparing a source register of one of said plurality of instructions to a destination register of each instruction in said instruction buffer that precedes said one of said plurality of instructions in the prescribed program order.

16. The superscalar processor of claim 15, wherein said data dependency checker comprises a plurality of data dependency circuits, wherein each dependency circuit performs at least on comparison to determine whether any given one of said plurality of instructions depends on a previous instruction.

17. The superscalar processor of claim 16, wherein one of said data dependency circuits comprises at least one comparator circuit, said comparator circuit receiving a first input

and a second input and outputting a dependency signal, said first input corresponding to a source register of a first instruction stored in said instruction buffer and said second input corresponding to a destination register of a second instruction stored in said instruction buffer, wherein said second instruction precedes said first instruction in the prescribed program order, and wherein said dependency signal indicates whether said first instruction is dependent on said second instruction.

18. The superscalar processor of claim **17**, wherein said comparator circuit comprises a plurality of exclusive NOR gates (XNOR) and an AND gate having a plurality of inputs, wherein an output of each XNOR gate is tied to an input of said AND gate, and wherein each XNOR has a first XNOR input and a second XNOR input, wherein each first XNOR input is tied to a bit of said first input and each second XNOR input is tied to a bit of said second input.

19. A computer system, comprising:

a memory unit for storing program instructions having a prescribed program order;

a bus for retrieving said program instructions from said memory unit; and

a processor in communication with said bus for executing said program instructions, wherein said processor comprises:

an instruction buffer for storing a plurality of instructions;

an index-addressable temporary buffer comprising a plurality of temporary storage locations, wherein each one of said plurality of instructions is assigned to a unique one of said plurality of temporary storage locations, wherein an output corresponding to a given one of said plurality of instructions is stored in said temporary storage location assigned to said given one of said plurality of instructions;

a data dependency checker to locate a dependent instruction stored in said instruction buffer, wherein said dependent instruction has an input that is dependent on a previous instruction, wherein said previous instruction is an instruction in said instruction buffer that precedes said dependent instruction in the prescribed program order; and

a circuit that receives from said data dependency checker dependency data corresponding to said dependent instruction and uses said dependency data to associate said temporary storage location assigned to said previous instruction with said input.

20. The computer system of claim **19**, wherein said circuit associates said temporary storage location assigned to said previous instruction with said input by outputting an address of said temporary storage location assigned to said previous instruction.

21. The computer system of claim **20**, wherein said circuit further outputs a first signal indicating a completion status of said previous instruction.

22. The computer system of claim **21**, further comprising:

a multiplexer having a first plurality of inputs, wherein said multiplexer receives an address of a storage location at each one of said plurality of inputs, wherein one of said addresses received at one of said plurality of inputs is said address outputted by said circuit; and

an issuer that receives as an input said signal indicating a completion status of said previous instruction, and that outputs a select signal to said multiplexer.

23. The computer system of claim **19**, wherein said circuit comprises a priority encoder and a multiplexer, wherein said encoder receives said dependency data, and an output of said encoder is used as a select signal for said multiplexer.

24. The computer system of claim **23**, wherein said output of said encoder represents said previous instruction.

25. The computer system of claim **24**, wherein said multiplexer receives a plurality of references as inputs and outputs a reference corresponding to said output of said encoder, wherein said reference outputted by said multiplexer represents an address of said temporary storage location assigned to said previous instruction.

26. The computer system of claim **19**, wherein said instruction buffer is capable of storing at most X number of instructions, and said temporary buffer includes at least X number of temporary storage locations, wherein X is a positive integer.

27. The computer system of claim **19**, wherein one of said plurality of said program instructions is assigned to a unique one of said plurality of storage locations based on a position of said one of said plurality of said program instructions within said instruction buffer.

28. The computer system of claim **19**, wherein said data dependency checker locates a dependent instruction stored in said instruction buffer by comparing a source register of one of said plurality of said program instructions to a destination register of each instruction in said instruction buffer that precedes said one of said plurality of said program instructions in the prescribed program order.

29. The computer system of claim **28**, wherein said data dependency checker comprises a plurality of data dependency circuits, wherein each dependency circuit performs at least on comparison to determine whether any given one of said plurality of said program instructions depends on a previous instruction.

30. The computer system of claim **29**, wherein one of said data dependency circuits comprises at least one comparator circuit, said comparator circuit receiving a first input and a second input and outputting a dependency signal, said first input corresponding to a source register of a first instruction stored in said instruction buffer and said second input corresponding to a destination register of a second instruction stored in said instruction buffer, wherein said second instruction precedes said first instruction in the prescribed program order, and wherein said dependency signal indicates whether said first instruction is dependent on said second instruction.

31. The computer system of claim **30**, wherein said comparator circuit comprises a plurality of exclusive NOR gates (XNOR) and an AND gate having a plurality of inputs, wherein an output of each XNOR gate is tied to an input of said AND gate, and wherein each XNOR has a first XNOR input and a second XNOR input, wherein each first XNOR input is tied to a bit of said first input and each second XNOR input is tied to a bit of said second input.

32. A superscalar processor for executing instructions having a prescribed program order, comprising:

an instruction buffer storing a plurality of instructions;

a register file having a plurality of registers and a plurality of index-addressable temporary storage locations, wherein each one of said plurality of instructions is assigned to a unique one of said plurality of temporary storage locations and one of said plurality of registers, wherein an output corresponding to a given one of said plurality of instructions is stored in said temporary storage location assigned to said given one of said plurality of instructions;

a data dependency checker to locate dependent instructions stored in said instruction buffer, wherein a dependent instruction is an instruction that should not be executed until after a particular previous instruction

within said instruction buffer is executed, wherein said data dependency checker outputs dependency information corresponding to each instruction stored in said instruction buffer;

a plurality of circuits, wherein each of said plurality of circuits corresponds to an instruction in said instruction buffer and receives dependency data from said data dependency checker, wherein said dependency data received at a particular circuit corresponds to said instruction to which said particular circuit corresponds, and wherein a circuit corresponding to a given dependent instruction outputs a reference representing said temporary storage location assigned to a particular previous instruction that must be executed prior to said given dependent instruction.

33. A computer system, comprising:

a memory unit for storing program instructions having a prescribed program order;

a bus for retrieving said program instructions from said memory unit; and

a processor in communication with said bus for executing said program instructions, wherein said processor comprises:

an instruction buffer storing a plurality of instructions;

a register file having a plurality of registers and a plurality of index-addressable temporary storage locations, wherein each one of said plurality of instructions is assigned to a unique one of said plurality of temporary storage locations and one of said plurality of registers, wherein an output corresponding to a given one of said plurality of instructions is stored in said temporary storage location assigned to said given one of said plurality of instructions;

a data dependency checker to locate dependent instructions stored in said instruction buffer, wherein a dependent instruction is an instruction that should not be executed until after a particular previous instruction within said instruction buffer is executed, wherein said data dependency checker outputs dependency information corresponding to each instruction stored in said instruction buffer; and

a plurality of circuits, wherein each of said plurality of circuits corresponds to an instruction in said instruction buffer and receives dependency data from said data dependency checker, wherein said dependency data received at a particular circuit corresponds to said instruction to which said particular circuit corresponds, and wherein a circuit corresponding to a given dependent instruction outputs a reference representing said temporary storage location assigned to a particular previous instruction that must be executed prior to said given dependent instruction.

34. In a computer system having a register file comprising a plurality of registers and a plurality of index-addressable temporary storage locations, a method for executing instructions having an input and an output and having a prescribed program order, comprising the steps of:

(1) assigning a unique one of the plurality of index-addressable temporary storage locations to each one of a plurality of instructions in an instruction buffer, wherein the output corresponding to a given one of said plurality of instructions is stored in said temporary storage location assigned to said given one of said plurality of instructions;

(2) determining whether one of said plurality of instructions in said instruction buffer is a dependent instruction, wherein said dependent instruction has an input that is dependent on a previous instruction, wherein said previous instruction is an instruction in said instruction buffer that precedes said dependent instruction in the prescribed program order; and

(3) associating said temporary storage location assigned to said previous instruction with the input that is dependent on said previous instruction.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 5,974,526
DATED : October 26, 1999
INVENTOR(S) : Garg et al.

Page 1 of 3

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Title page,
Item [75], Inventors, please replace "**Le Trong Nguyen's**" address "Sereno" with -- Monte Sereno --.
Item [73], Assignee, please replace "**Seiko Corporation**" with -- **Seiko Epson Corporation** --.
Item [57], **ABSTRACT**,
Line 8, please replace "of" with -- or --.

Drawings,
Please replace Figures 6A and 6B (sheets 6 and 7) with the attached Figures 6A and 6B.

Column 10,
Line 12, please replace "I9D" with -- I0D --.
Line 16, please replace "T0D" with -- I0D --.

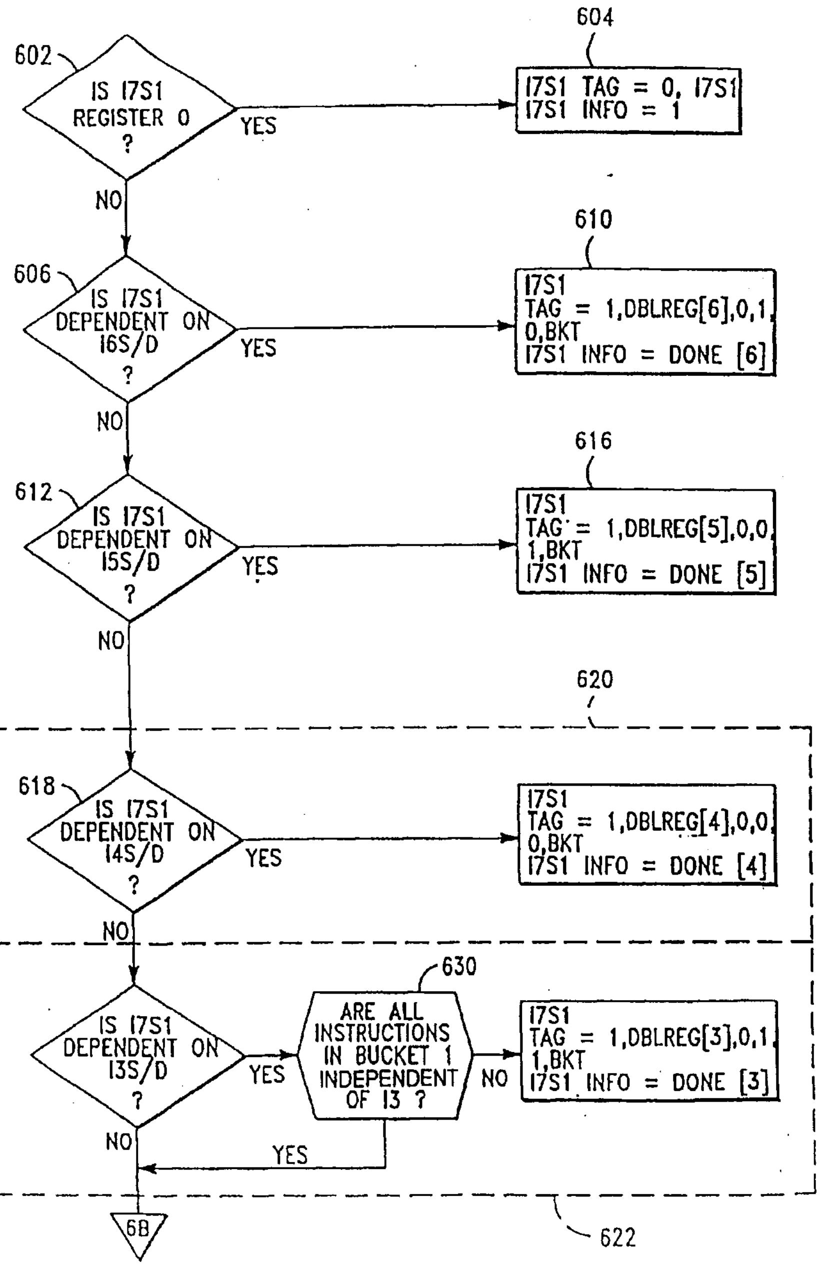
Column 12,
Line 60, first occurrence, please replace "r2" with -- r1 --.
Line 67, please replace "IOS/D" with -- I0S/D --.

Column 13,
Line 1, please replace "IOS/D" with -- I0S/D --.
Line 11, please replace "t0" with -- r0 --.
Line 20, please replace "IOS/D" with -- I0S/D --, and please replace "12SD" with -- I2SD --.
Line 24, please replace "I15/D" with -- I1S/D --.
Between lines 36 and 44, please replace "IO" with -- I0 --, 4 occurrences.

Signed and Sealed this

Twenty-fifth Day of November, 2003

JAMES E. ROGAN
Director of the United States Patent and Trademark Office



602
IS I7S1 REGISTER 0 ?
YES
604
I7S1 TAG = 0, I7S1 I7S1 INFO = 1
NO
606
IS I7S1 DEPENDENT ON I6S/D ?
YES
610
I7S1 TAG = 1,DBLREG[6],0,1, 0,BKT I7S1 INFO = DONE [6]
NO
612
IS I7S1 DEPENDENT ON I5S/D ?
YES
616
I7S1 TAG = 1,DBLREG[5],0,0, 1,BKT I7S1 INFO = DONE [5]
NO
620
618
IS I7S1 DEPENDENT ON I4S/D ?
YES
I7S1 TAG = 1,DBLREG[4],0,0, 0,BKT I7S1 INFO = DONE [4]
NO
630
IS I7S1 DEPENDENT ON I3S/D ?
YES
ARE ALL INSTRUCTIONS IN BUCKET 1 INDEPENDENT OF I3 ?
NO
I7S1 TAG = 1,DBLREG[3],0,1, 1,BKT I7S1 INFO = DONE [3]
NO
YES
622
6B

FIG.—6A

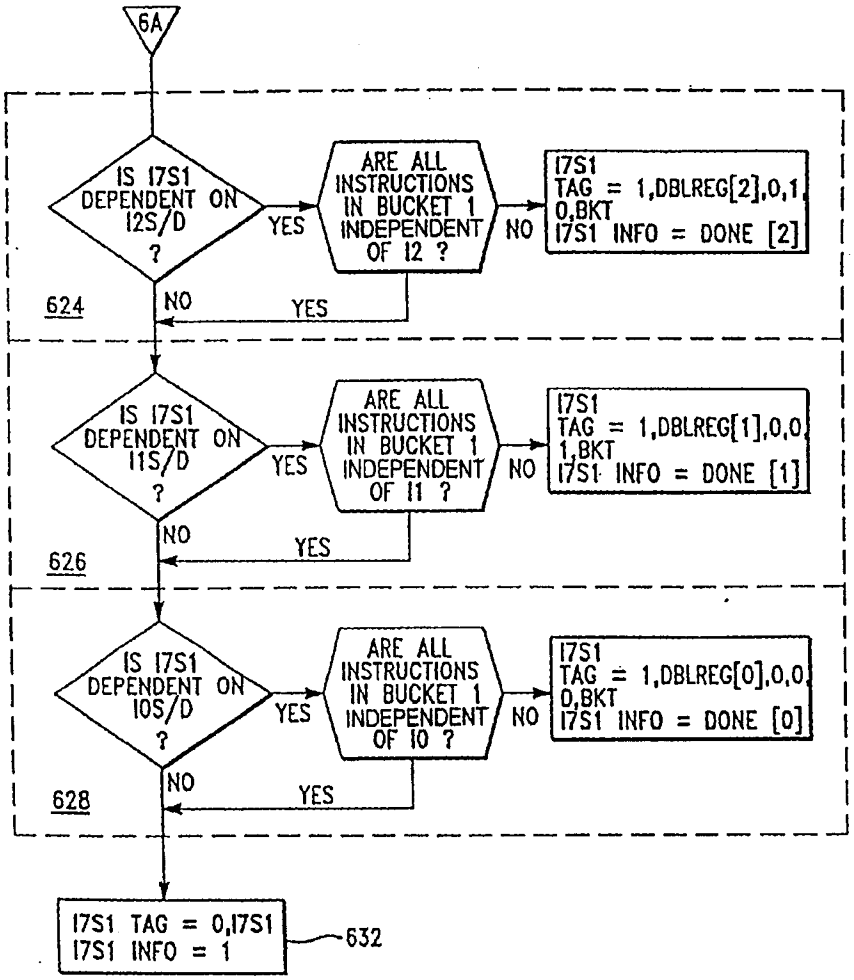

6A
IS I7S1 DEPENDENT ON I2S/D ?
YES
ARE ALL INSTRUCTIONS IN BUCKET 1 INDEPENDENT OF I2 ?
NO
I7S1 TAG = 1,DBLREG[2],0,1, 0,BKT I7S1 INFO = DONE [2]
624
NO
YES
IS I7S1 DEPENDENT ON I1S/D ?
YES
ARE ALL INSTRUCTIONS IN BUCKET 1 INDEPENDENT OF I1 ?
NO
I7S1 TAG = 1,DBLREG[1],0,0, 1,BKT I7S1 INFO = DONE [1]
626
NO
YES
IS I7S1 DEPENDENT ON I0S/D ?
YES
ARE ALL INSTRUCTIONS IN BUCKET 1 INDEPENDENT OF I0 ?
NO
I7S1 TAG = 1,DBLREG[0],0,0, 0,BKT I7S1 INFO = DONE [0]
628
NO
YES
I7S1 TAG = 0,I7S1 I7S1 INFO = 1
632

FIG.—6B

EXHIBIT J

US006289433B1

(12) **United States Patent**
Garg et al.

(10) **Patent No.: US 6,289,433 B1**
(45) **Date of Patent: *Sep. 11, 2001**

(54) **SUPERSCALAR RISC INSTRUCTION SCHEDULING**

(75) Inventors: **Sanjiv Garg**, Freemont; **Kevin Ray Iadonato**, San Jose; **Le Trong Nguyen**, Monte Sereno; **Johannes Wang**, Redwood City, all of CA (US)

(73) Assignee: **Transmeta Corporation**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: **09/329,354**

(22) Filed: **Jun. 10, 1999**

Related U.S. Application Data

(63) Continuation of application No. 08/990,414, filed on Dec. 15, 1997, now Pat. No. 5,974,526, which is a continuation-in-part of application No. 08/594,401, filed on Jan. 31, 1996, now Pat. No. 5,737,624, which is a continuation of application No. 08/219,425, filed on Mar. 29, 1994, now Pat. No. 5,497,499, which is a continuation of application No. 07/860,719, filed on Mar. 31, 1992, now abandoned.

(51) **Int. Cl.**[7] **G06F 15/00**
(52) **U.S. Cl.** **712/23**; 712/218; 712/216
(58) **Field of Search** 712/23, 24, 216, 712/217, 218, 215

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,626,989	12/1986	Torii	364/200
4,675,806	6/1987	Uchida	364/200
4,722,049	1/1988	Lahti	364/200
4,807,115	2/1989	Torng	364/200
4,901,233	2/1990	Liptay	395/375
4,903,196	2/1990	Pomerene et al.	364/200
4,942,525	7/1990	Shintani et al.	395/375
4,992,938	2/1991	Cocke et al.	364/200

(List continued on next page.)

FOREIGN PATENT DOCUMENTS

0 515 166	11/1992	(EP) .
0 533 337	3/1993	(EP) .
WO 91/20031	12/1991	(WO) .

OTHER PUBLICATIONS

Acosta, Ramón D. et al., "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors," *IEEE Transactions On Computers*, vol. C–35, No. 9, Sep. 1986, pp. 815–828.

(List continued on next page.)

Wilson, J.E. et al., "On Turning the Microarchitecture of an HPS Implementation of the VAX," *Proceedings of Micro 20*, Dec. 1987, pp. 162–167.

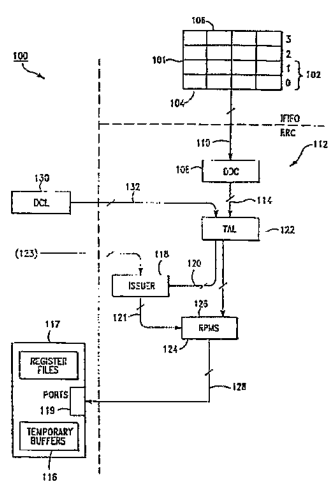
IBM Journal of Research and Development, vol. 34, No. 1, Jan. 1990, pp. 1–70.

Primary Examiner—Larry D. Donaghue
(74) *Attorney, Agent, or Firm*—Sterne, Kessler, Goldstein & Fox, P.L.L.C.

(57) **ABSTRACT**

A register renaming system for out-of-order execution of a set of reduced instruction set computer instructions having addressable source and destination register fields, adapted for use in a computer having an instruction execution unit with a register file accessed by read address ports and for storing instruction operands. A data dependence check circuit is included for determining data dependencies between the instructions. A tag assignment circuit generates one of more tags to specify the location of operands, based on the data dependencies determined by the data dependence check circuit. A set of register file port multiplexers select the tags generated by the tag assignment circuit and pass the tags onto the read address ports of the register file for storing execution results.

19 Claims, 9 Drawing Sheets



U.S. PATENT DOCUMENTS

5,067,069	11/1991	Fite et al.	395/375
5,109,495	4/1992	Fite et al.	395/375
5,142,633	8/1992	Murray et al.	395/375
5,214,763	5/1993	Blaner et al.	395/375
5,222,244	6/1993	Carbine et al.	395/800
5,226,126	7/1993	McFarland et al.	395/375
5,230,068	7/1993	Van Dyke et al.	395/375
5,251,306	10/1993	Tran	395/375
5,261,071	11/1993	Lyon	395/425
5,345,569	9/1994	Tran	395/375
5,355,457	10/1994	Shebanow et al.	395/375
5,398,330	3/1995	Johnson	395/375
5,442,757	8/1995	McFarland et al.	395/375
5,448,705	9/1995	Nguyen et al.	395/375
5,487,156	1/1996	Popescu et al.	395/375
5,497,499	3/1996	Garg et al.	395/800.73
5,561,776	10/1996	Popescu et al.	395/375
5,574,927	11/1996	Scantlin	395/800
5,592,636	1/1997	Popescu et al.	395/586
5,625,837	4/1997	Popescu et al.	395/800
5,627,983	5/1997	Popescu et al.	395/393
5,708,841	1/1998	Popescu et al.	355/800
5,737,624	4/1998	Garg et al.	395/800.73
5,768,575	6/1998	McFarland et al.	395/569
5,778,210	7/1998	Henstrom et al.	395/394
5,797,025	8/1998	Popescu et al.	395/800
5,832,205	11/1998	Kelly et al.	395/185.06
5,832,293	11/1998	Popescu et al.	395/800.23

OTHER PUBLICATIONS

Agerwala et al., "High Performance Reduced Instruction Set Processors," IBM Research Division, Mar. 31, 1987, pp. 1–61.

Aiken, A. and Nicolau, A., "Perfect Pipelining: A New Loop Parallelization Technique*," pp. 221–235.

Butler, M. and Patt, Y., "An Improved Area–Efficient Register Alias Table for Implementing HPS," University of Michigan, Ann Arbor, Michigan, Jan. 1990, pp. 1–15.

Butler, M. et al., "Single Instruction Stream Parallelism Is Greater Than Two," *Proceedings of ISCA–18,* May 1990, pp. 276–286.

Charlesworth, A.E., "An Approach to Scientific Array Processing: The Architectural Design of the AP–120B/FPS–164 Family," *Computer,* vol. 14, Sep. 1981, pp. 18–27.

Colwell, et al., "A VLIW Architecture for a Trace Scheduling Compiler," Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 1987, pp. 180–192.

Dywer, "A Multiple, Out–of–Order, Instruction Issuing System For Superscaler Processors," (All); Aug. 1991.

Foster et al., "Percolation of Code to Enhance Parallel Dispatching and Execution," *IEEE Trans. On Computers,* Dec. 1971, pp. 1411–1415.

Gee, J. et al., "The Implementation of Prolog via VAX 8600 Microcode," *Proceedings of Micro 19,* New York City, Oct. 1986, pp. 1–7.

Goodman, J.R. and Hsu, W., "Code Scheduling and Register Allocation in Large Basic Blocks," *ACM,* 1988, pp. 442–452.

Gross et al., "Optimizing Delayed Branches," Proceedings of the 5th Annual Workshop on Microprogramming, Oct. 5–7, 1982, pp. 114–120.

Groves, R.D. and Oehler, R., "An IBM Second Generation RISC Processor Architecture," IEEE, 1989, pp. 134–137.

Hennessy, J.L and Patterson, D.A., *Computer Architecture A Quantitative Approach,* 1990, Ch. 6.4, 6.7 and p. 449.

Horst, R.W. et al., "Multiple Instruction Issue in the NonStop Cyclone Processor," IEEE, 1990, pp. 216–226.

Hwu, W. et al., "An HPS Implementation of VAX: Initial Design and Analysis," *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences,* 1986, pp. 282–291.

Hwu, W. et al., "Checkpoint Repair for High–Performance Out–of–Order Execution Machines," *IEEE Trans. On Computers,* vol. C–36, No. 12, Dec. 1987, pp. 1496–1514.

Hwu, W. and Patt, Y.N., "Design Choices for the HPSm Microprocessor Chip," *Proceedings of the Twentieth Annual Hawaii International Conference on System Sciences,* 1987, pp. 330–336, 1987.

Hwu, W. et al., "Experiments with HPS, a Restricted Data Flow Microarchitecture for High Performance Computers," *COMPCON 86,* 1986.

Hwu, W. et al., "Exploiting Parallel Microprocessor Microarchitectures with a Compiler Code Generator," *Proceedings of the 15th Annual Symposium on Computer Architecture,* Jun. 1988, pp. 45–53.

Hwu, W. and Patt, Y.N., "HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality," *Proceedings of the 18th International Symposium on Computer Architecture,* pp. 297–306, Jun. 1986.

Hwu, W. and Patt, Y.N., "HPSm2: A Refined Single–chip Microengine," *HICSS '88,* 1988, pp. 30–40.

Johnson, William M., *Super–Scalar Processor Design,* (Dissertation), Copyright 1989, 134 pages.

Jouppi et al., "Available Instruction–Level Parallelism for Superscalar and Superpipelined Machines," *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems,* Apr. 1989, pp. 272–282.

Jouppi, N.H., "Integration and Packaging Plateaus of Processor Performance," IEEE, 1989, pp. 229–232.

Jouppi, N.P., "The Nonuniform Distribution of Instruction–Level and Machine Parallelism and Its Effect on Performance," *IEEE Transactions on Computers,* vol. 38, No. 12, Dec. 1989, pp. 1645–1658.

Keller, "Look–Ahead Processors"; Dec. 1975, pp. 177–194.

Lam, M.S., "Instruction Scheduling For Superscalar Architectures," *Annu. Rev. Comput. Sci.,* vol. 4, 1990, pp. 173–201.

Lightner et al., "The Metaflow Architecture", *IEEE Micro Magazine,* Jun. 1991, pp. 11–12 and 63–68.

Lightner et al., "The Metaflow Lightning" Chip Set Mar. 1991 *IEEE Lightning Outlined. Microprocesor Report.* Sep. 1990.

Melvin, S. and Patt, Y., "Exploiting Fine–Grained Parallelism Through a Combination of Hardware and Software Techniques," *Proceedings From ISCA–18,* May 1990, pp. 287–296.

Murakami, K. et al., "SIMP (Single Instruction stream/ Multiple instruction Pipelining): A Novel High–Speed Single–Processor Architecture," *ACM,* 1989, pp. 78–85.

Patt, Y.N. et al., "Critical Issues Regarding HPS, A High Performance Microarchitecture," The 18th Annual Workshop on Microprogramming, Pacfiic Grove, California, Dec. 3–6, 1985, IEEE Computer Order No. 653, pp. 109–116.

Patt, Y.N. et al., "HPS, A New Microarchitecture: Rationale and Introduction," The 18th Annual Workshop on Microprogramming, Pacific Grove, California, Dec. 3–6, 1985; IEEE Computer Society Order No. 653, pp. 103–108.

Patt, Y.N. et al., "Run–Time Generation of HPS Microinstructions From a VAX Instruction Stream," *Proceedings of MICRO 19 Workshop,* New York, New York, Oct. 1986, pp. 1–7.

Peleg et al., "Future Trends in Microprocessors: Out–of–Order Execution, Spec. Branching and Their CISC Performance Potential", Mar. 1991.

Pleszkun et al., "The Performance Potential of Multiple Functional Unit Processors," *Proceedings of the 15th Annual Symposium on Computer Architecture,* Jun. 1988, pp. 37–44.

Pleszkun et al., "WISQ: A Restartable Architecture Using Queues," *Proceedings of the 14th International Symposium on Computer Architecture,* Jun. 1987, pp. 290–299.

Smith, M.D. et al., "Boosting Beyond Static Scheduling in a Superscalar Processor," IEEE, 1990, pp. 344–354.

Smith, et al., "Implementation of Precise Interrupts in Pipelined Processors," *Proceedings of the 12th Annual International Symposium on Computer Architecture,* Jun. 1985, pp. 36–44.

Smith, M.D. et al., "Limits on Multiple Instruction Issue," *Computer Architecture News,* No. 2, Apr. 17, 1989, pp. 290–302.

Sohi, G.S. et al., "Instruction Issue Logic for High Performance, Interruptable Pipelined Processors," The 14th Annual International Symposium on Computer Architecture, Jun. 2–5, 1987, pp. 27–34.

Swenson, J.A. and Patt, Y.N., "Hierarchical Registers for Scientific Computers," St. Malo '88, University of California at Berkeley, 1988, pp. 346–353.

Thornton, J.E., *Design of a Computer: The Control Data 6600,* Control Data Corporation, 1970, pp. 58–140.

Tjaden et al., "Detection and Parallel Execution of Independent Instructions," *IEEE Trans. On Computers,* vol. C–19, No. 10, Oct. 1970, pp. 889–895.

Tjaden, et al., "Representation of Concurrency with Ordering Matrices," *IEEE Trans. On Computers,* vol. C–22, No. 8, Aug. 1973, pp. 752–761.

Tjaden, *Representation and Detection of Concurrency Using Ordering Matrices,* (Dissertation), 1972, pp. 1–199.

Tomasulo, R.M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal,* vol. 11, Jan. 1967, pp. 25–33.

Uht, A.K., "An Efficient Hardware Algorithm to Extract Concurrency From General–Purpose Code," *Proceedings of the 19th Annual Hawaii International Conference on System Sciences,* 1986, pp. 41–50.

Uvieghara, G.A. et al., "An Experimental Single–Chip Data Flow CPU," *IEEE Journal of Solid–State Circuits,* vol. 27, No. 1, Jan. 1992, pp. 17–28.

Uvieghara, G.A. et al., "An Experimental Single–Chip Data Flow CPU," Symposium on ULSI Circuits Design Digest of Technical Papers, May 1990.

Wedig, R.G., *Detection of Concurrency In Directly Executed Language Instruction Streams,* (Dissertation), Jun. 1982, pp. 1–179.

Weiss et al., "Instruction Issue Logic in Pipelined Supercomputers," Reprinted from *IEEE Trans. on Computers,* vol. C–33, No. 11, Nov. 1984, pp. 1013–1022.

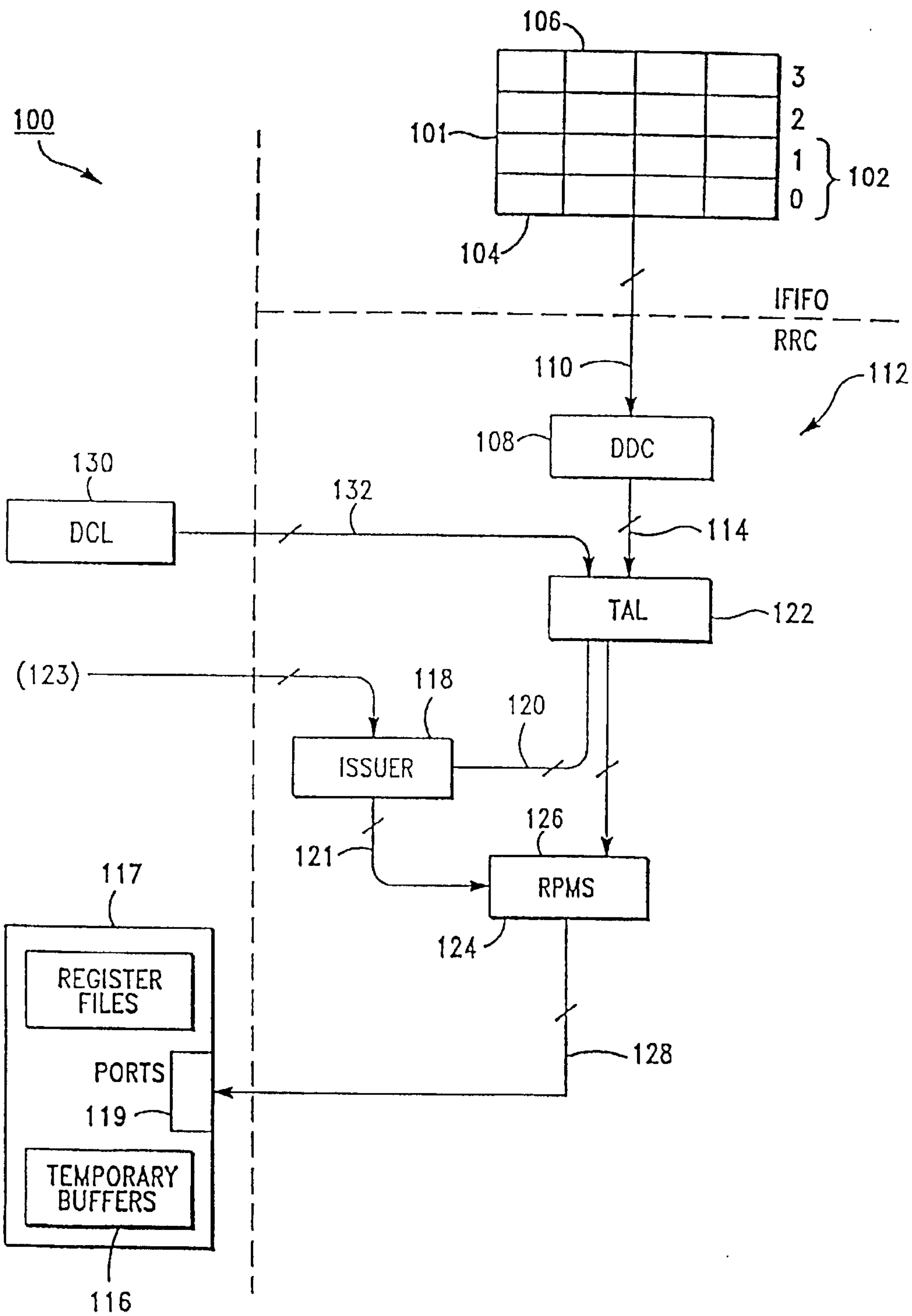
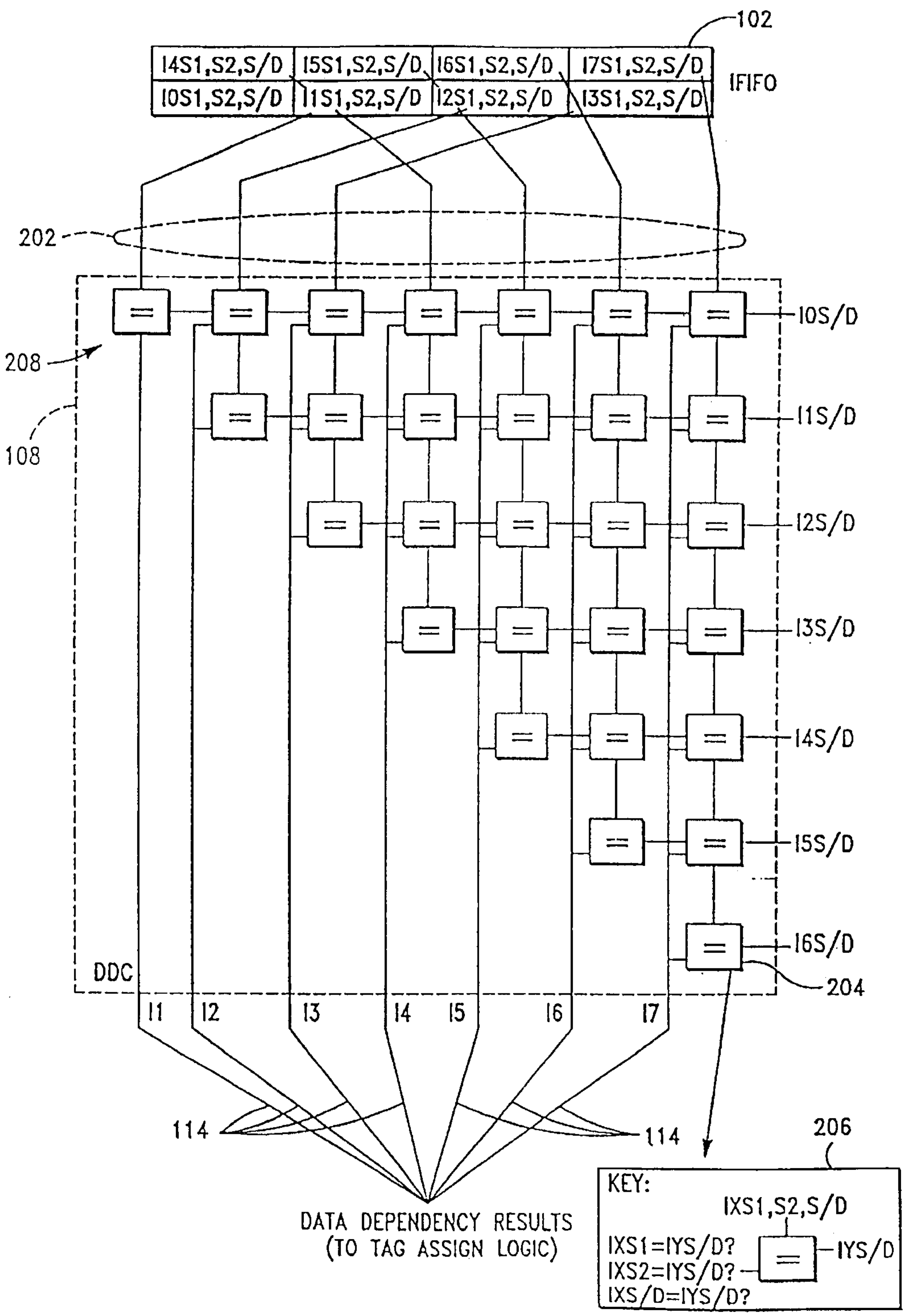
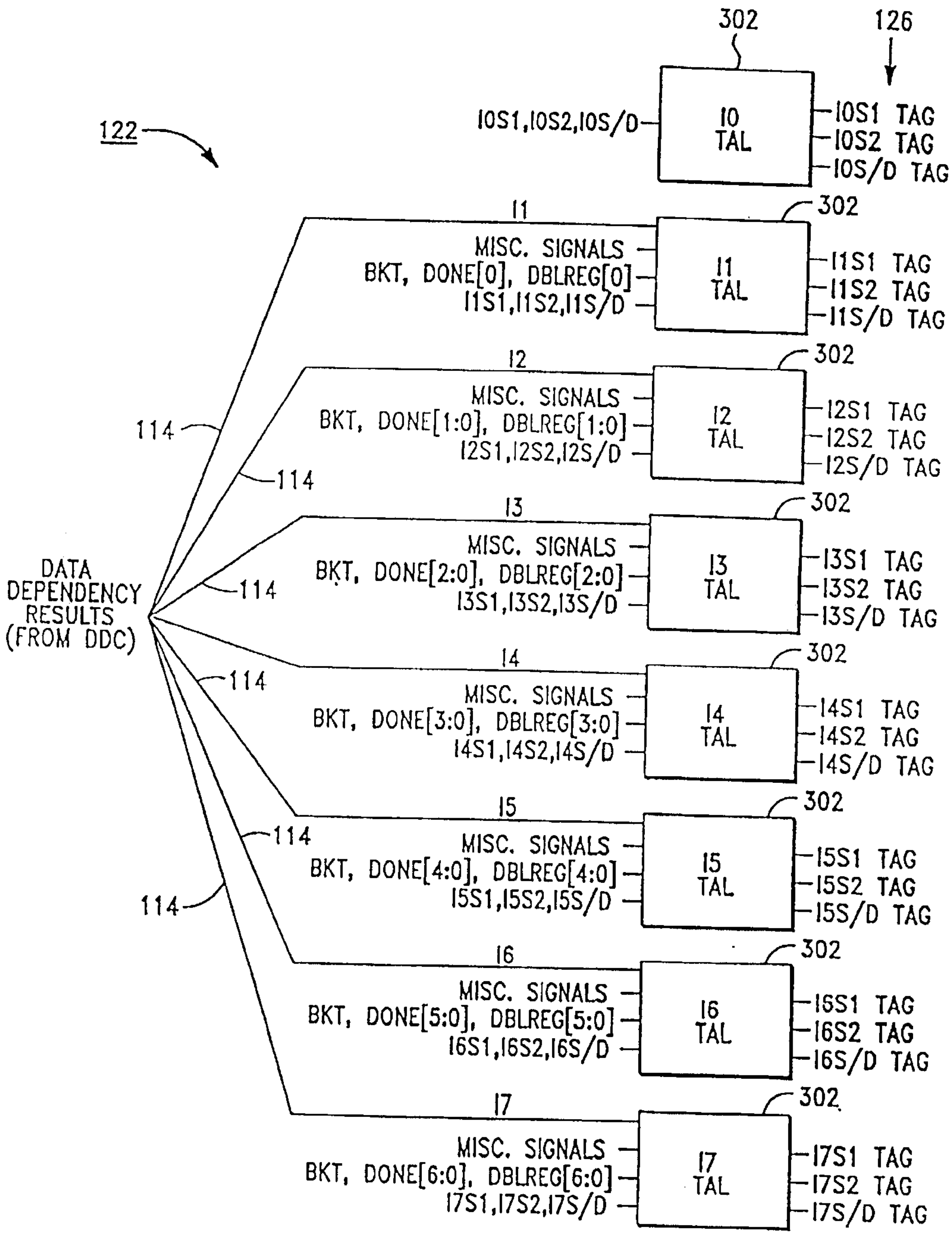

100
106
101
3
2
1
0
102
104
IFIFO
RRC
110
112
108
DDC
130
DCL
132
114
TAL
122
(123)
118
120
ISSUER
126
121
RPMS
124
117
REGISTER FILES
PORTS
119
128
TEMPORARY BUFFERS
116

FIG.—1



102
I4S1,S2,S/D
I5S1,S2,S/D
I6S1,S2,S/D
I7S1,S2,S/D
I0S1,S2,S/D
I1S1,S2,S/D
I2S1,S2,S/D
I3S1,S2,S/D
IFIFO
202
208
108
I0S/D
I1S/D
I2S/D
I3S/D
I4S/D
I5S/D
I6S/D
DDC
204
I1
I2
I3
I4
I5
I6
I7
114
114
206
KEY:
IXS1,S2,S/D
IXS1=IYS/D?
IXS2=IYS/D?
IXS/D=IYS/D?
IYS/D
DATA DEPENDENCY RESULTS
(TO TAG ASSIGN LOGIC)

FIG.—2



122
302
126
I0S1,I0S2,I0S/D
I0 TAL
I0S1 TAG
I0S2 TAG
I0S/D TAG
DATA DEPENDENCY RESULTS (FROM DDC)
114
I1
MISC. SIGNALS
BKT, DONE[0], DBLREG[0]
I1S1,I1S2,I1S/D
I1 TAL
I1S1 TAG
I1S2 TAG
I1S/D TAG
I2
MISC. SIGNALS
BKT, DONE[1:0], DBLREG[1:0]
I2S1,I2S2,I2S/D
I2 TAL
I2S1 TAG
I2S2 TAG
I2S/D TAG
I3
MISC. SIGNALS
BKT, DONE[2:0], DBLREG[2:0]
I3S1,I3S2,I3S/D
I3 TAL
I3S1 TAG
I3S2 TAG
I3S/D TAG
I4
MISC. SIGNALS
BKT, DONE[3:0], DBLREG[3:0]
I4S1,I4S2,I4S/D
I4 TAL
I4S1 TAG
I4S2 TAG
I4S/D TAG
I5
MISC. SIGNALS
BKT, DONE[4:0], DBLREG[4:0]
I5S1,I5S2,I5S/D
I5 TAL
I5S1 TAG
I5S2 TAG
I5S/D TAG
I6
MISC. SIGNALS
BKT, DONE[5:0], DBLREG[5:0]
I6S1,I6S2,I6S/D
I6 TAL
I6S1 TAG
I6S2 TAG
I6S/D TAG
I7
MISC. SIGNALS
BKT, DONE[6:0], DBLREG[6:0]
I7S1,I7S2,I7S/D
I7 TAL
I7S1 TAG
I7S2 TAG
I7S/D TAG

FIG.—3

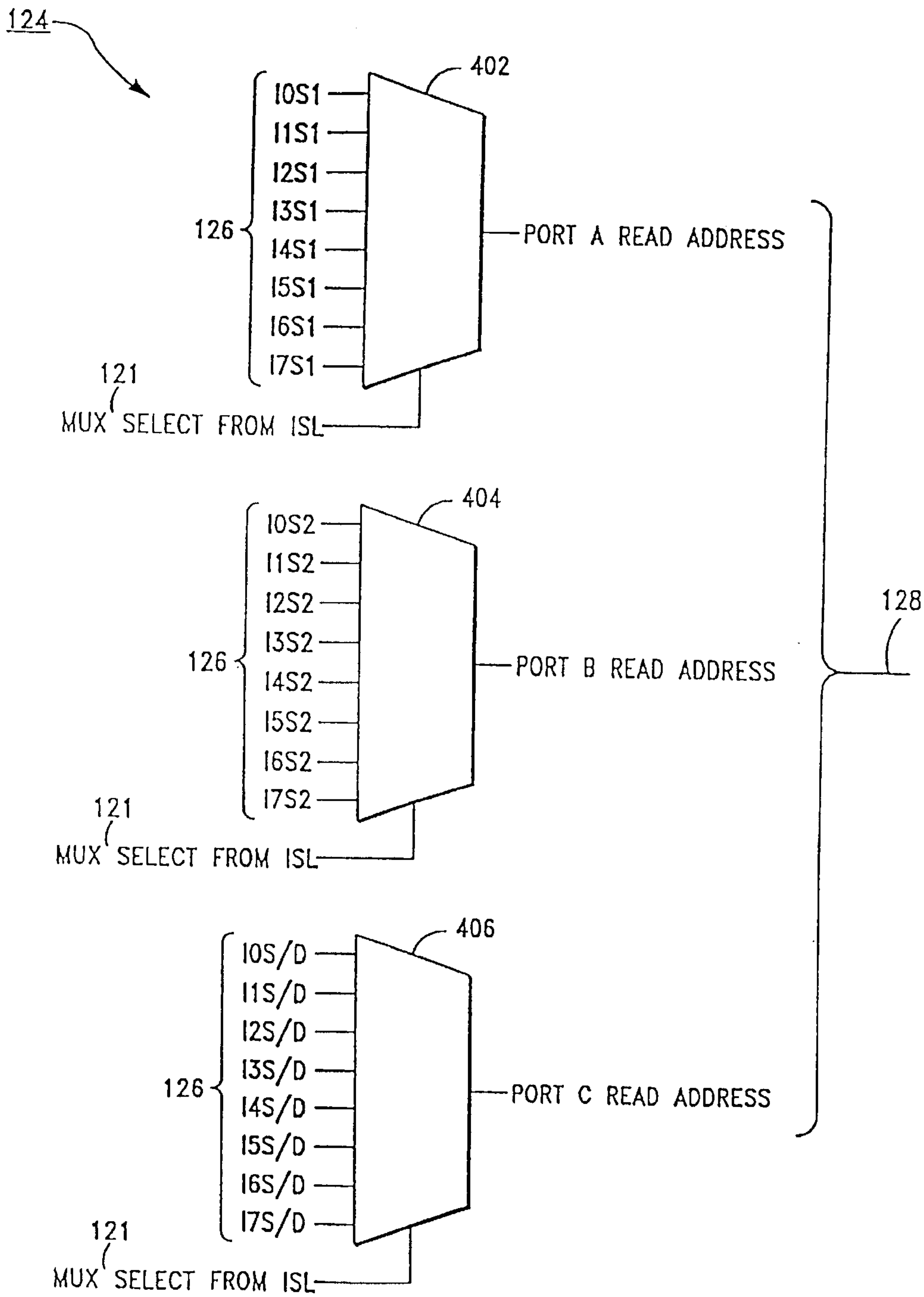
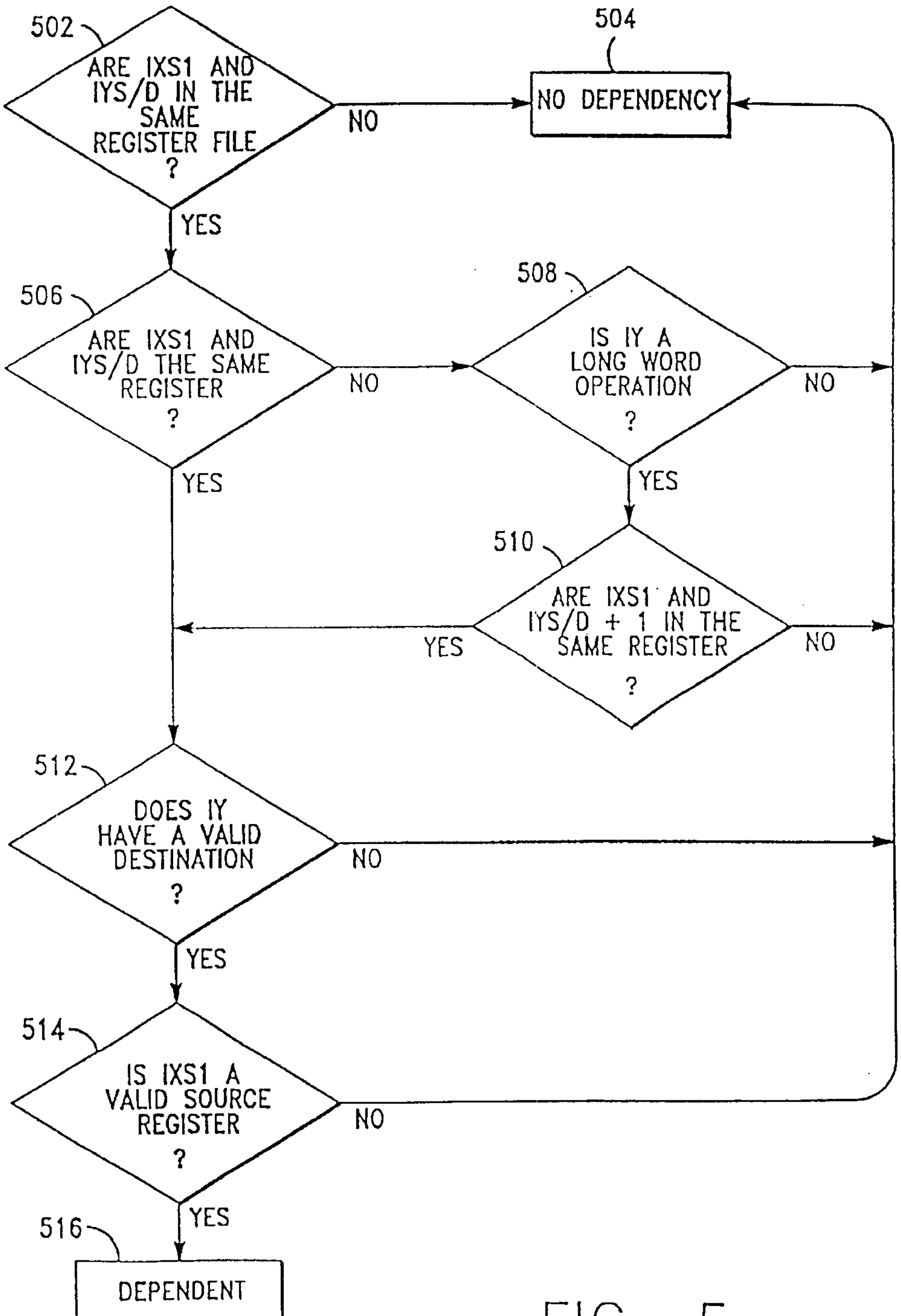

124
402
I0S1
I1S1
I2S1
I3S1
I4S1
I5S1
I6S1
I7S1
126
PORT A READ ADDRESS
121
MUX SELECT FROM ISL
404
I0S2
I1S2
I2S2
I3S2
I4S2
I5S2
I6S2
I7S2
126
PORT B READ ADDRESS
128
121
MUX SELECT FROM ISL
406
I0S/D
I1S/D
I2S/D
I3S/D
I4S/D
I5S/D
I6S/D
I7S/D
126
PORT C READ ADDRESS
121
MUX SELECT FROM ISL

FIG.—4



502
ARE IXS1 AND IYS/D IN THE SAME REGISTER FILE ?
NO
504
NO DEPENDENCY
YES
506
ARE IXS1 AND IYS/D THE SAME REGISTER ?
NO
508
IS IY A LONG WORD OPERATION ?
NO
YES
YES
510
ARE IXS1 AND IYS/D + 1 IN THE SAME REGISTER ?
YES
NO
512
DOES IY HAVE A VALID DESTINATION ?
NO
YES
514
IS IXS1 A VALID SOURCE REGISTER ?
NO
YES
516
DEPENDENT

FIG.-5

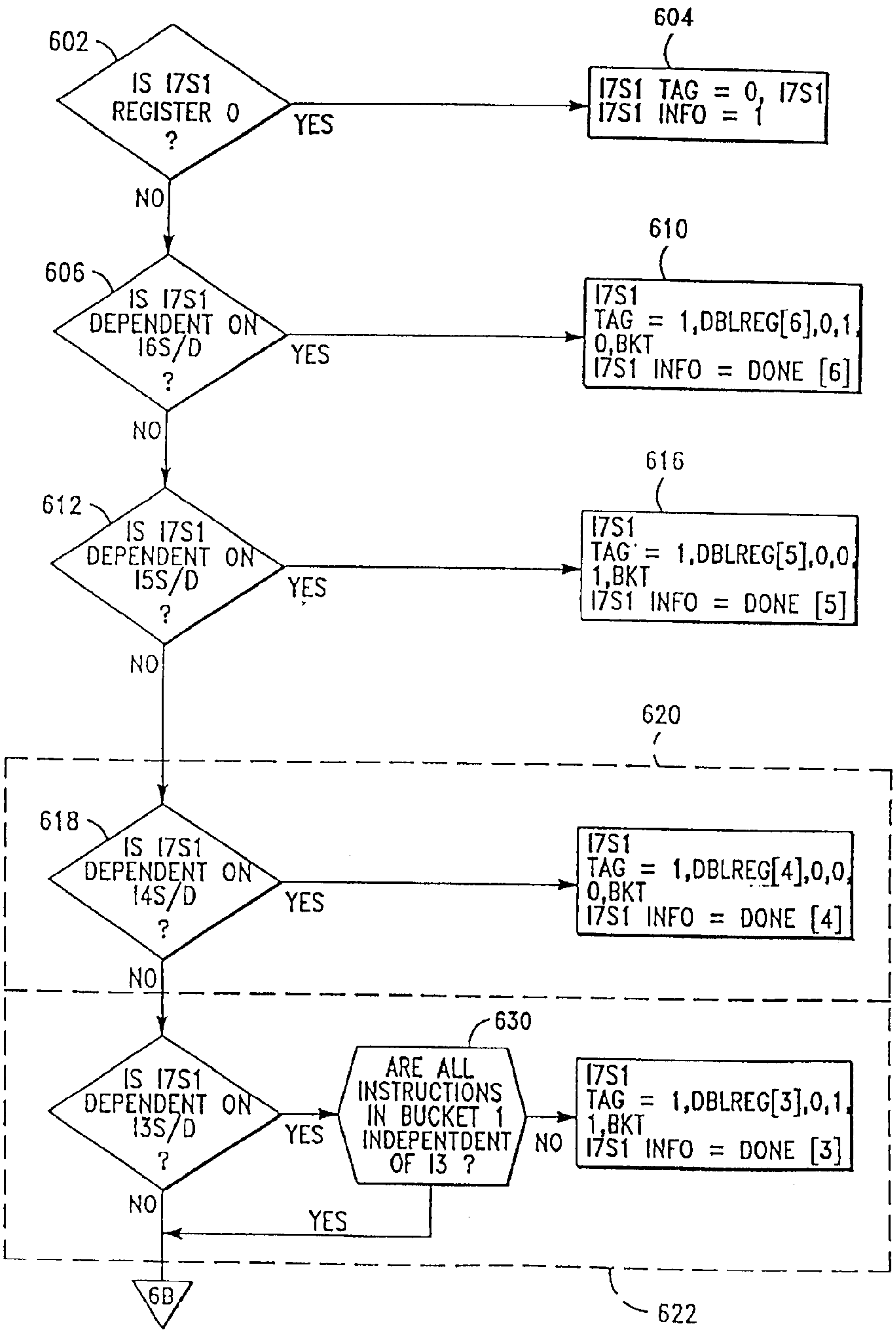
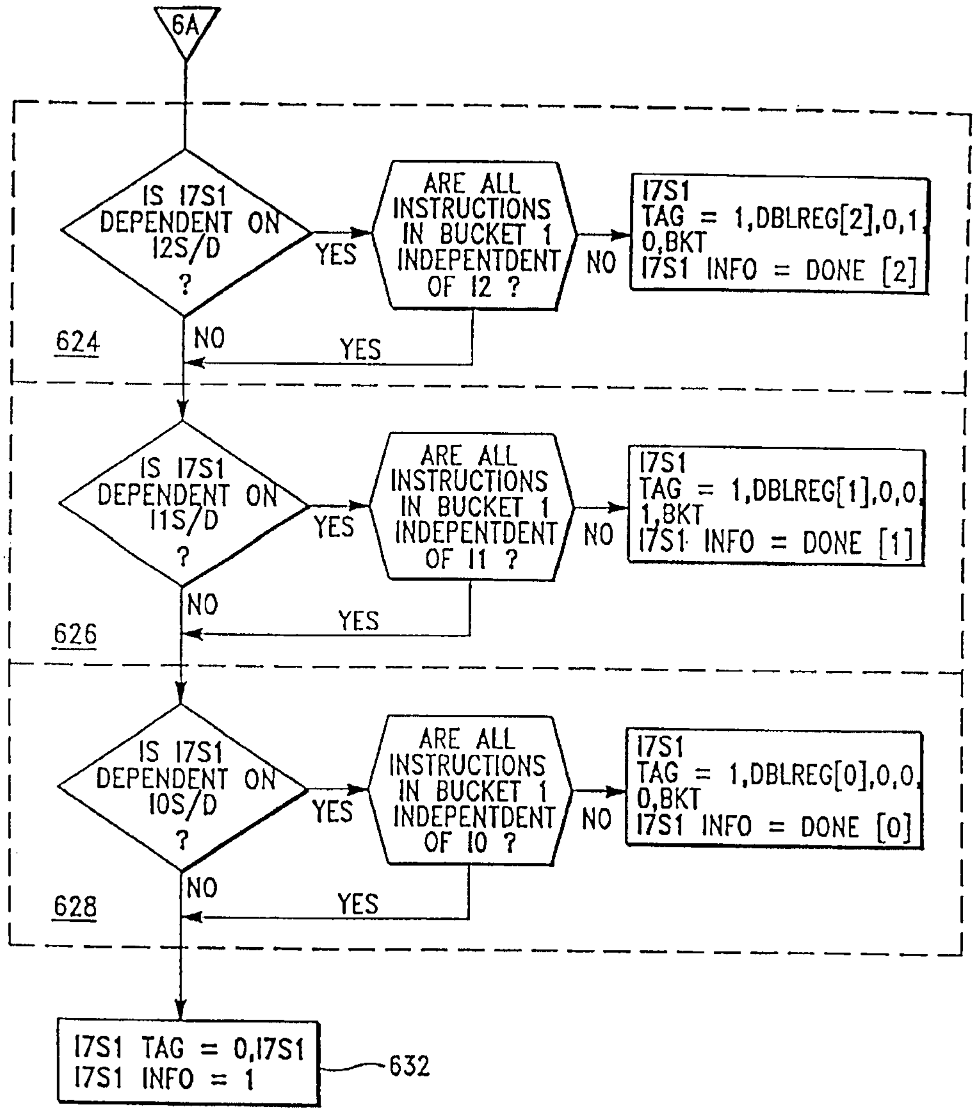

602
IS I7S1 REGISTER 0 ?
YES
604
I7S1 TAG = 0, I7S1
I7S1 INFO = 1
NO
606
IS I7S1 DEPENDENT ON I6S/D ?
YES
610
I7S1
TAG = 1,DBLREG[6],0,1,
0,BKT
I7S1 INFO = DONE [6]
NO
612
IS I7S1 DEPENDENT ON I5S/D ?
YES
616
I7S1
TAG = 1,DBLREG[5],0,0,
1,BKT
I7S1 INFO = DONE [5]
NO
620
618
IS I7S1 DEPENDENT ON I4S/D ?
YES
I7S1
TAG = 1,DBLREG[4],0,0,
0,BKT
I7S1 INFO = DONE [4]
NO
630
IS I7S1 DEPENDENT ON I3S/D ?
YES
ARE ALL INSTRUCTIONS IN BUCKET 1 INDEPENTDENT OF I3 ?
NO
I7S1
TAG = 1,DBLREG[3],0,1,
1,BKT
I7S1 INFO = DONE [3]
NO
YES
6B
622

FIG.—6A



6A
IS I7S1 DEPENDENT ON I2S/D ?
YES
ARE ALL INSTRUCTIONS IN BUCKET 1 INDEPENTDENT OF I2 ?
NO
I7S1 TAG = 1,DBLREG[2],0,1, 0,BKT I7S1 INFO = DONE [2]
624
NO
YES
IS I7S1 DEPENDENT ON I1S/D ?
YES
ARE ALL INSTRUCTIONS IN BUCKET 1 INDEPENTDENT OF I1 ?
NO
I7S1 TAG = 1,DBLREG[1],0,0, 1,BKT I7S1 INFO = DONE [1]
626
NO
YES
IS I7S1 DEPENDENT ON I0S/D ?
YES
ARE ALL INSTRUCTIONS IN BUCKET 1 INDEPENTDENT OF I0 ?
NO
I7S1 TAG = 1,DBLREG[0],0,0, 0,BKT I7S1 INFO = DONE [0]
628
NO
YES
I7S1 TAG = 0,I7S1 I7S1 INFO = 1
632

FIG.—6B

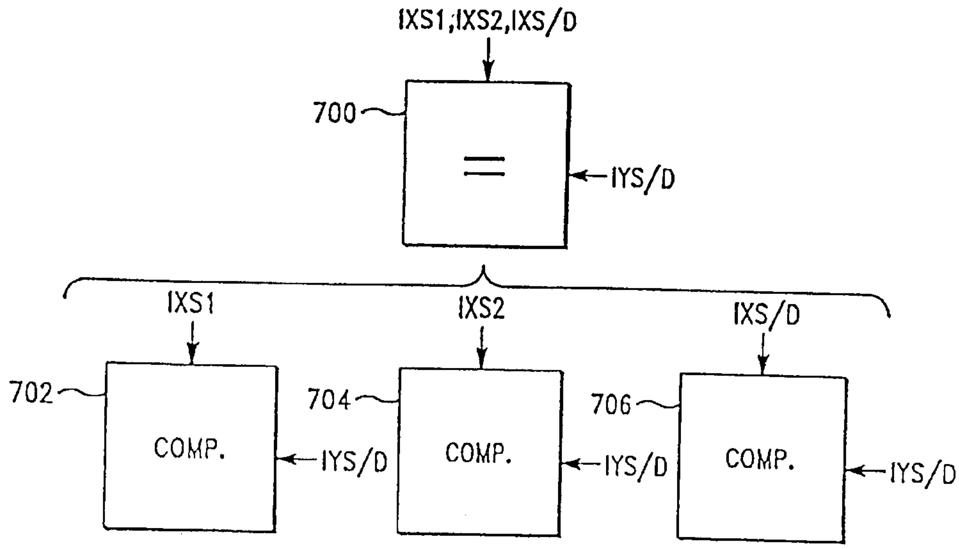

IXS1,IXS2,IXS/D
700
=
IYS/D
IXS1
IXS2
IXS/D
702
COMP.
704
IYS/D
COMP.
706
IYS/D
COMP.
IYS/D

FIG.—7

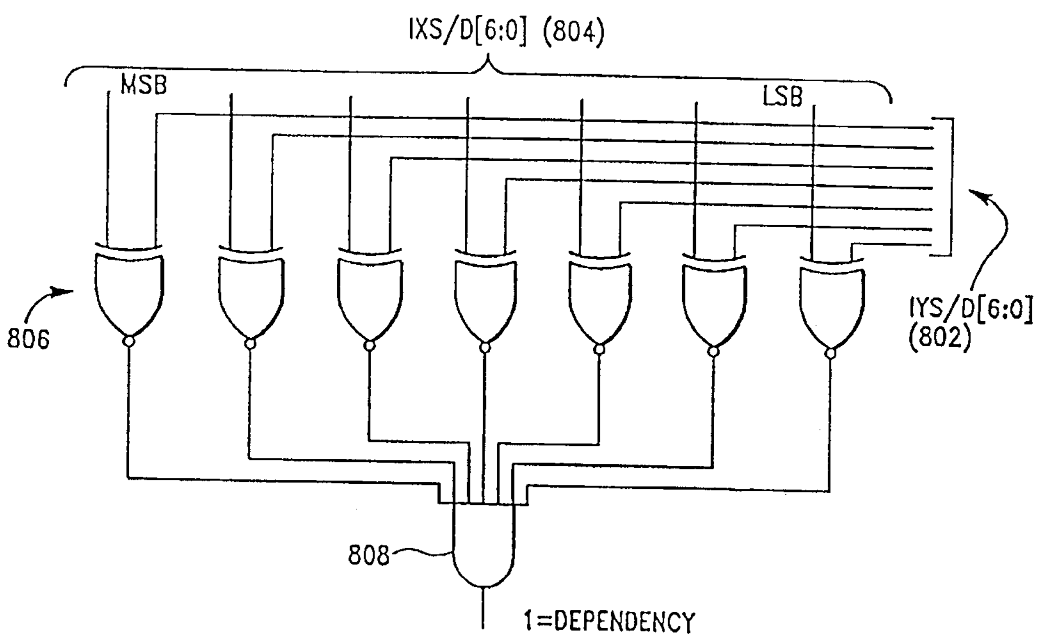

IXS/D[6:0] (804)
MSB
LSB
806
IYS/D[6:0]
(802)
808
1=DEPENDENCY

FIG.—8

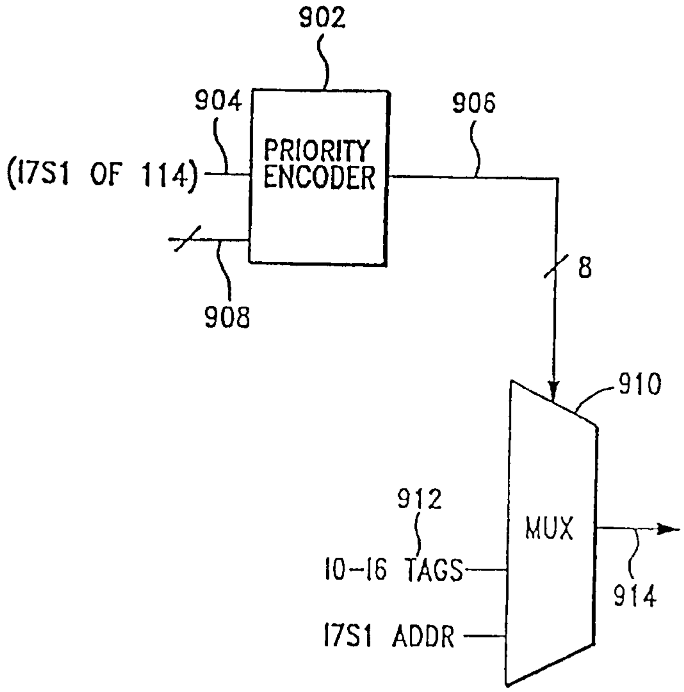

902
904
906
PRIORITY ENCODER
(17S1 OF 114)
8
908
910
912
MUX
10-16 TAGS
914
17S1 ADDR

FIG.-9

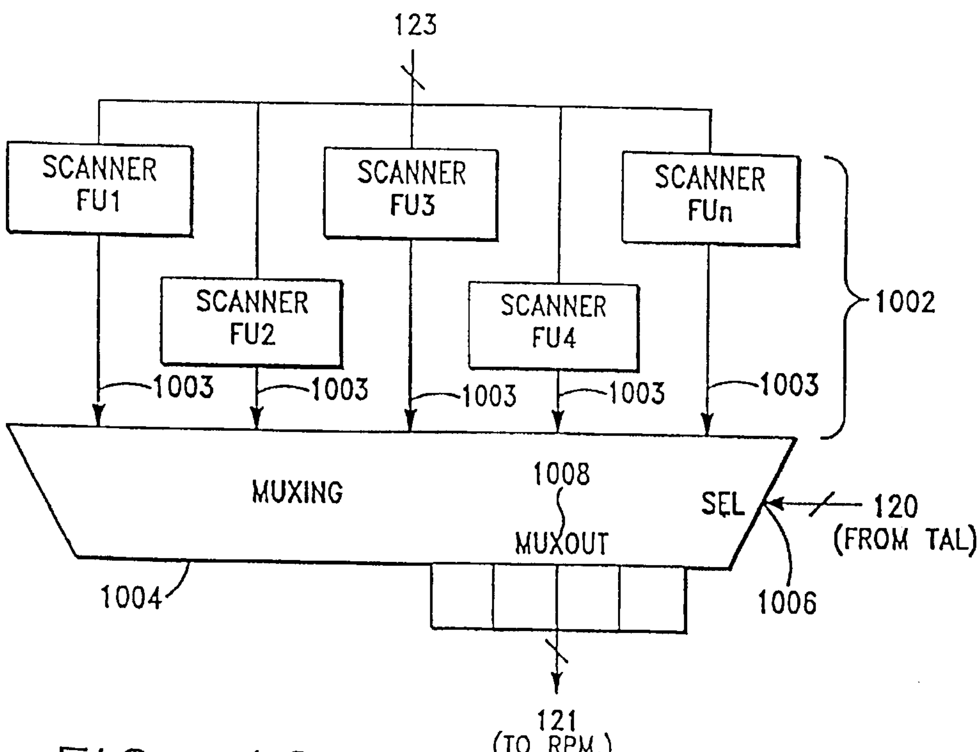

123
SCANNER FU1
SCANNER FU2
SCANNER FU3
SCANNER FU4
SCANNER FUn
1002
1003
1003
1003
1003
1003
MUXING
1008
SEL
120
(FROM TAL)
MUXOUT
1004
1006
121
(TO RPM.)

FIG.-10

SUPERSCALAR RISC INSTRUCTION SCHEDULING

This application is a continuation of appl. Ser. No. 08/990,414, filed Dec. 15, 1997, now U.S. Pat. No. 5,974, 526, which is a continuation of appl. Ser. No. 08/594,401, filed Jan. 31, 1996, now U.S. Pat. No. 5,737,624, which is a continuation of appl. Ser. No. 08/219,425, filed Mar. 29, 1994, now U.S. Pat. No. 5,497,499 which is a continuation of appl. Ser. No. 07/860,719, filed Mar. 31, 1992 (status: abandoned).

CROSS-REFERENCE TO RELATED APPLICATIONS

The following are commonly owned applications:

"Semiconductor Floor Plan and Method for a Register Renaming Circuit", Ser. No. 04/860,718, concurrently filed with the present application, now U.S. Pat. No. 5,371,684;

"High Performance RISC Microprocessor Architecture", Ser. No. 07/817,810, filed Jan. 8, 1992 now U.S. Pat. No. 5,539,911;

"Extensible RISC Microprocessor Architecture", Ser. No. 07/817,809, filed Jan. 8, 1992, now abandoned.

The disclosures of the above applications are incorporated herein by reference.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to superscalar reduced instruction set computers (RISC), more particularly, the present invention relates to instruction scheduling including register renaming and instruction issuing for superscalar RISC computers.

2. Related Art

A more detailed description of some of the basic concepts discussed in this application is found in a number of references, including Mike Johnson, Superscalar Microprocessor Design (Prentice-Hall, Inc., Englewood Ciffs, N.J., 1991); John L. Hennessy et. al., Computer Architecture—A Quantitative Approach" (Morgan Kaufmann Publishers, Inc., San Mateo, Calif., 1990). Johnson's text, particularly Chapters 2, 6 and 7 provide an excellent discussion of the register renaming issues addressed by the present invention.

A major consideration in a superscalar RISC processor is to how to execute multiple instructions in parallel and out-of-order, without incurring data errors due to dependencies inherent in such execution. Data dependency checking, register renaming and instruction scheduling are integral aspects of the solution.

Storage Conflicts and Register Renaming

True dependencies (sometimes called "flow dependencies" or "write-read" dependencies) are often grouped with anti-dependencies (also called "read-write" dependencies) and output dependencies (also called "write-write" dependencies) into a single group of instruction dependencies. The reason for this grouping is that each of these dependencies manifests itself through use of registers or other storage locations. However, it is important to distinguish true dependencies from the other two. True dependencies represent the flow of data and information through a program. Anti- and output dependencies arise because, at different points in time, registers or other storage locations hold different values for different computations.

When instructions are issued in order and complete in order, there is a one-to-one correspondence between registers and values. At any given point in execution, a register identifier precisely identifies the value contained in the corresponding register. When instructions are issued out of order and complete out of order, correspondence between registers and values breaks down, and values conflict for registers. This problem is severe when the goal of register allocation is to keep as many values in as few registers as possible. Keeping a large number of values in a small number of registers creates a large number of conflicts when the execution order is changed from the order assumed by the register allocator.

Anti- and output dependencies are more properly called "storage conflicts" because reusing storage locations (including registers) causes instructions to interfere with one another even though conflicting instructions are otherwise independent. Storage conflicts constrain instruction issue and reduce performance. But storage conflicts, like other resource conflicts, can be reduced or eliminated by duplicating the troublesome resource.

Dependency Mechanisms

Johnson also discusses in detail various dependency mechanisms, including: software, register renaming, register renaming with a reorder buffer, register renaming with a future buffer, interlocks, the copying of operands in the instruction window to avoid dependencies, and partial renaming.

A conventional hardware implementation relies on software to enforce dependencies between instructions. A compiler or other code generator can arrange the order of instructions so that the hardware cannot possibly see an instruction until it is free of true dependencies and storage conflicts. Unfortunately, this approach runs into several problems. Software does not always know the latency of processor operations, and thus, cannot always know how to arrange instructions to avoid dependencies. There is the question of how the software prevents the hardware from seeing an instruction until it is free of dependencies. In a scalar processor with low operation latencies, software can insert "no-ops" in the code to satisfy data dependencies without too much overhead. If the processor is attempting to fetch several instructions per cycle, or if some operations take several cycles to complete, the number of no-ops required to prevent the processor from seeing dependent instructions rapidly becomes excessive, causing an unacceptable increase in code size. The no-ops use a precious resource, the instruction cache, to encode dependencies between instructions.

When a processor permits out-of-order issue, it is not at all clear what mechanism software should use to enforce dependencies. Software has little control over the behavior of the processor, so it is hard to see how software prevents the processor from decoding dependent instructions. The second consideration is that no existing binary code for any scalar processor enforces the dependencies in a superscalar processor, because the mode of execution is very different in the superscalar processor. Relying on software to enforce dependencies requires that the code be regenerated for the superscalar processor. Finally, the dependencies in the code are directly determined by the latencies in the hardware, so that the best code for each version of a superscalar processor depends on the implementation of that version.

On the other hand, there is some motivation against hardware dependency techniques, because they are inher-

ently complex. Assuming instructions with two input operands and one output value, as holds for typical RISC instructions, then there are five possible dependencies between any two instructions: two true dependencies, two anti-dependencies, and one output dependency. Furthermore, the number of dependencies between a group of instructions, such as a group of instructions in a window, varies with the square of the number of instructions in the group, because each instruction must be considered against every other instruction.

Complexity is further multiplied by the number of instructions that the processor attempts to decode, issue, and complete in a single cycle. These actions introduce dependencies. The only aid in reducing complexity is that the dependencies can be determined incrementally, over many cycles to help reduce the scope and complexity of the dependency hardware.

One technique for removing storage conflicts is by providing additional registers that are used to reestablish the correspondence between registers and values. The additional registers are conventionally allocated dynamically by hardware, and the registers are associated with values needed by the program using "register renaming." To implement register renaming, processors typically allocate a new register for every new value produced (i.e., for every instruction that writes a register). An instruction identifying the original register, for the purpose of reading its value, obtains instead the value in the newly allocated register. Thus, hardware renames the original register identifier in the instruction to identify the new register and correct value. The same register identifier in several different instructions may access different hardware registers, depending on the locations of register references with respect to register assignments.

Consider the following code sequence where "op" is an operation, "Rn" represents a numbered register, and ":=" represents assignment:

$$R3b:=R3a \text{ op } R5a \quad (1)$$

$$R4b:=R3b+1 \quad (2)$$

$$R3c:=R5a+1 \quad (3)$$

$$R7b:=R3c \text{ op } R4b \quad (4)$$

Each assignment to a register creates a new "instance" of the register, denoted by an alphabetic subscript. The creation of a new instance for R3 in the third instruction avoids the anti- and output dependencies on the second and first instructions, respectively, and yet does not interfere with correctly supplying an operand to the fourth instruction. The assignment to R3 in the third instruction supersedes the assignment to R3 in the first instruction, causing R3*c* to become the new R3 seen by subsequent instructions until another instruction assigns a value to R3.

Hardware that performs renaming creates each new register instance and destroys the instance when its value is superseded and there are no outstanding references to the value. This removes anti- and output dependencies and allows more instruction parallelism. Registers are still reused, but reuse is in line with the requirements of parallel execution. This is particularly helpful with out-of-order issue, because storage conflicts introduce instruction issue constraints that are not really necessary to produce correct results. For example, in the preceding instruction sequence, renaming allows the third instruction to be issued immediately, whereas, without renaming, the instruction

must be delayed until the first instruction is complete and the second instruction is issued.

Another technique for reducing dependencies is to associate a single bit (called a "scoreboard bit") with each register. The scoreboard bit is used to indicate that a register has a pending update. When an instruction is decoded that will write a register, the processor sets the associated scoreboard bit. The scoreboard bit is reset when the write actually occurs. Because there is only one scoreboard bit indicating whether or not there is a pending update, there can be only one such update for each register. The scoreboard stalls instruction decoding if a decoded instruction will update a register that already has a pending update (indicated by the scoreboard bit being set). This avoids output dependencies by allowing only one pending update to a register at any given time.

Register renaming, in contrast, uses multiple-bit tags to identify the various uncomputed values, some of which values may be destined for the same processor register (that is, the same program-visible register). Conventional renaming requires hardware to allocate tags from a pool of available tags that are not currently associated with any value and requires hardware to free the tags to the pool once the values have been computed. Furthermore, since scoreboarding allows only one pending update to a given register, the processor is not concerned about which update is the most recent.

A further technique for reducing dependencies is using register renaming with a "reorder buffer" which uses associative lookup. The associative lookup maps the register identifier to the reorder buffer entry as soon as the entry is allocated, and, to avoid output dependencies, the lookup is prioritized so that only the value for the most recent assignment is obtained if the register is assigned more than once. A tag is obtained if the result is not yet available. There can be as many instances of a given register as there are reorder buffer entries, so there are no storage conflicts between instructions. The values for the different instances are written from the reorder buffer to the register file in sequential order. When the value for the final instance is written to the register file, the reorder buffer no longer maps the register, the register file contains the only instance of the register, and this is the most recent instance.

However, renaming with a reorder buffer relies on the associative lookup in the reorder buffer to map register identifiers to values. In the reorder buffer, the associative lookup is prioritized so that the reorder buffer always provides the most recent value in the register of interest (or a tag). The reorder buffer also writes values to the register file in order, so that, if the value is not in the reorder buffer, the register file must contain the most recent value.

In a still further technique for reducing dependencies, associative lookup can be eliminated using a "future file." The future file does not have the properties of the reorder buffer discussed in the preceding paragraph. A value presented to the future file to be written may not be the most recent value destined for the corresponding register, and the value cannot be treated as the most recent value unless it actually is. The future file therefore keeps track of the most recent update and checks that each write corresponds to the most recent update before it actually performs the write.

When an instruction is decoded, it accesses tags in the future file along with the operand values. If the register has one or more pending updates, the tag identifies the update value required by the decoded instruction. Once an instruction is decoded, other instructions may overwrite this instructions's source operands without being constrained by

anti-dependencies, because the operands are copied into the instruction window. Output dependencies are handled by preventing the writing as a result into the future file if the result does not have a tag for the most recent value. Both anti- and output dependencies are handled without stalling instruction issue.

If dependencies are not removed through renaming, "interlocks" must use to enforce dependencies. An interlock simply delays the execution of an instruction until the instruction is free of dependencies. There are two ways to prevent an instruction from being executed: one way is to prevent the instruction from being decoded, and the other is to prevent the instruction from being issued.

To improve performance over scoreboarding, interlocks are moved from the decoder to the instruction window using a "dispatch stack." The dispatch stack is an instruction window that augments each instruction in the window with dependency counts. There is a dependency count associated with the source register of each instruction in the window, giving the number of pending prior updates to the source register and thus the number of updates that must be completed before all possible true dependencies are removed. There are two similar dependency counts associated with the destination register of each instruction in the window, giving both the number of pending prior uses of the register (which is the number of anti-dependencies) and the number of pending prior updates to the register (which is the number of output dependencies).

When an instruction is decoded and loaded into the dispatch stack, the dependency counts are set by comparing the instruction's register identifiers with the register identifiers of all instructions already in the dispatch stack. As instructions complete, the dependency counts of instructions that are still in the window are decremented based on the source and destination register identifiers of completing instructions (the counts are decremented by a variable amount, depending on the number of instructions completed). An instruction is independent when all of its counts are zero. The use of counts avoids having to compare all instructions in the dispatch stack to all other instructions on every cycle.

Anti-dependencies can be avoided altogether by copying operands to the instruction window (for example, to the reservation stations) during instruction decode. In this manner, the operands cannot be overwritten by subsequent register updates. Operands can be copied to eliminate anti-dependencies in any approach, independent of register renaming. The alternative to copying operands is to interlock anti-dependencies, but the comparators and/or counters required for these interlocks are costly, considering the number of combinations of source and result registers to be compared.

A tag can be supplied for the operand rather than the operand itself. This tag is simply a means for the hardware to identify which value the instruction requires, so that, when the operand value is produced, it can be matched to the instruction. If there can be only one pending update to a register, the register identifier can serve as a tag (as with scoreboarding). If there can be more than one pending update to a register (as with renaming), there must be a mechanism for allocating result tags and insuring uniqueness.

An alternative to scoreboarding interlocking is to allow multiple pending updates of registers to avoid stalling the decoder for output dependencies, but to handle anti-dependencies by copying operands (or tags) during decode. An instruction in the window is not issued until it is free of

output dependencies, so the updates to each register are performed in the same order in which they would be performed with in-order completion, except that updates for different registers are out of order with respect to each other. This alternative has almost all of the capabilities of register renaming, lacking only the capability to issue instructions so that updates to the same register occur out of order.

There appears to be no better alternative to renaming other than with a reorder buffer. Underlying the discussion of dependencies has been the assumption that the processor performs out-of-order issue and already has a reorder buffer for recovering from mispredicted branches. Out-of-order issue makes it unacceptable to stall the decoder for dependencies. If the processor has an instruction window, it is inconsistent to limit the look ahead capability of the processor by interlocking the decoder. There are then only two alternatives: implement anti- and output dependency interlocks in the window or remove these altogether with renaming.

SUMMARY OF THE INVENTION

The present invention is directed to instruction scheduling including register renaming and instruction issuing for superscalar RISC computers. A Register Rename Circuit (RRC), which is part of the scheduling logic allows a computer's Instruction Execution Unit (IEU) to execute several instructions at the same time while avoiding dependencies. In contrast to conventional register renaming, the present invention does not actually rename register addresses. The RRC of the present invention temporarily buffers the instruction results, and the results of out-of-order instruction execution are not transferred to the register file until all previous instructions are done. The RRC also performs result forwarding to provide temporarily buffered operands (results) to dependant instructions. The RRC contains three subsections: a Data Dependency Checker (DDC), Tag Assign Logic (TAL) and Register file Port MUXes (RPM).

The function of the DDC is to locate the dependencies between the instructions for a group of instructions. The DDC does this by comparing the addresses of the source registers of each instruction to the addresses of the destination registers of each previous instruction in the group. For example, if instruction A reads a value from a register that is written to by instruction B, then instruction A is dependent upon instruction B and instruction A cannot start until instruction B has finished. The DDC outputs indicate these dependencies.

The outputs of the DDC go to the TAL. Because it is possible for an instruction to be dependent on more than one previous instruction, the TAL must determine which of those previous instructions will be the last one to be executed The present invention automatically maps each instruction a predetermined temporary buffer location; hence, the present invention does not need prioritized associative look-up as used by convention reorder buffers, thereby saving chip area/cost and execution speed.

Out-of-order results for several instructions being executed at the same time are stored in a set of temporary buffers, rather that the file register designated by the instruction. If the DDC determines, for example, that a register that instruction **6**'s source is written to by instructions **2**, **3** and **5**, then the TAL will indicate that instruction **6** must wait for instruction **5** by outputting the "tag" of instruction **5** for instruction **6**. The tag of instruction **5** shows the temporary buffer location where instruction **5**'s result is stored. It also

contains a one bit signal (called a "done flag") that indicates if instruction **5** is finished or not. The TAL will output three tags for each instruction, because each instruction can have three source registers. If an instruction is not dependent on any previous instruction, the TAL will output the register file address of the instruction's input, rather an a temporary buffer's address.

The last part of the RRC are the RPMs or Register file Port MUXes. The inputs of the RPMs are the outputs of the TAL, and the select lines for the RPMs come from another part of the IEU called the Instruction Scheduler or Issuer. The Instruction Scheduler chooses which instruction to execute (this decision is based partly on the done flags) and then uses the RPMs to select the tags of that instruction. These tags go to the read address ports of the computer's register files. In the previous example, once instruction **5** has finished, the Instruction Scheduler will start instruction **6**. It will select the RPM so that the address of instruction **5**'s result (its tag) is sent to the register file, and the register file will make the result of instruction **5** available to instruction **6**.

The foregoing and other features and advantages of the present invention will be apparent from the following more particular description of the preferred embodiments of the invention, as illustrated in the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention will be better understood if reference is made to the accompanying drawings.

FIG. **1** shows a representative high level block diagram of the register renaming circuit of the present invention.

FIG. **2** shows a representative block diagram of the data dependency check circuit of the present invention.

FIG. **3** shows a representative block diagram of the tag assignment logic of the present invention.

FIG. **4** shows a representative block diagram of the register port file multiplexers of the present invention.

FIG. **5** is a representative flowchart showing a data dependency check method for IXS1 and IYS/D in accordance with the present invention.

FIGS. **6**A and **6**B are representative flowcharts showing a tag assignment method in accordance with the present invention.

FIG. **7** shows a representative block diagram which compares an instruction Y's source/destination operand with each operand of an instruction X in accordance with an embodiment of the present invention.

FIG. **8** shows a representative circuit diagram for comparator block **706** of FIG. **7**.

FIG. **9** shows a representative block diagram of a Priority Encoder in accordance with an embodiment of the present invention.

FIG. **10** shows a representative block diagram of the instruction scheduling logic of the present invention.

DETAILED DESCRIPTION

FIG. **1** shows a representative high level block diagram of an Instruction Execution Unit (IEU) **100** associated with the present invention. The goal of IEU **100** is to execute as many instructions as possible in the shortest amount of time. There are two basic ways to accomplish this: optimize IEU **100** so that each instruction takes as little time as possible or optimize IEU **100** so that it can execute several instructions at the same time.

Instructions are sent to IEU **100** from an Instruction Fetch Unit (IFU, not shown) through an instruction FIFO (first-in-first-out register stack storage device) **101** in groups of four called "buckets." IEU **100** can decode and schedule up to two buckets of instructions at one time. FIFO **101** stores **16** total instructions in four buckets labeled **0–3**. IEU **100** looks at the an instruction window **102**. In one embodiment of the present invention, window **102** comprises eight instructions (buckets **0** and **1**). Every cycle IEU **100** tries to issue a maximum number of instructions from window **102**. Window **102** functions as a instruction buffer register. Once the instructions in a bucket are executed and their results stored in the processor's register file (see block **117**), the bucket is flushed out at a bottom **104** and a new bucket is dropped in at a top **106**.

In order to execute instructions in parallel or out of order, care must be taken so that the data that each instruction needs is available when the instruction needs it and also so that the result of each instruction is available for any future instructions that might need it. A Register Rename Circuit (RRC), which is part of the scheduling logic of the computer's IEU performs this function by locating dependencies between current instructions and then renaming the sources (inputs) of the instruction.

As noted above, there are three types of dependencies: input dependencies, output dependencies and anti-dependencies. Input dependencies occur when an instruction, call it A, that performs an operation on the result of a previous instruction, call it B. Output dependencies occur when the outputs of A and B are to be stored in the same place. Anti-dependencies occur when instruction A comes before B in the instruction stream and B's result will be stored in the same place as one of A's inputs.

Input dependencies are handled by not executing instructions until their inputs are available. RRC **112** is used to locate the input dependencies between current instructions and then to signal an Instruction Scheduler or Issuer **118** when all inputs for a particular instruction are ready. In order to locate these dependencies, RRC **112** compares the register file addresses of each instruction's inputs with the addresses of each previous instruction's output using a data dependency circuit (DDC) **108**. If one instruction's input comes from a register where a previous instruction's output will be stored, then the latter instruction must wait for the former to finish.

This implementation of RRC **112** can check eight instructions at the same time, so a current instruction is defined as any one of those eight from window **102**. It should become evident to those skilled in the art that the present invention can easily be adapted to check more or less instructions.

In one embodiment of the present invention, instructions can have from **0** to **3** inputs and **0** or **1** outputs. Most instructions' inputs and outputs come from, or are stored in, one of several register files. Each register file **117** (e.g., separate integer, floating and boolean register files) has 32 real entries plus the group of 8 temporary buffers **116**. When an instruction completes, (The term "complete" means that the operation is complete and the operand is ready to be written to its destination register.) its result is stored in its preassigned location in the temporary buffers **116**. Its result is later moved to the appropriate place in register file **117** after all previous instructions' results have been moved to their places in the register file. This movement of results from temporary buffers **116** to register file **117** is called "retirement" and is controlled by termination logic, as should become evident to those skilled in the art. More than one instruction may be retired at a time. Retirement comprises updating the "official state" of the machine including

the computer's Program Counter, as will become evident to those skilled in the art. For example, if instruction 10 happens to complete directly before instruction I1, both results can be stored directly into register file **117**. But if instruction I3 then completes, its result must be stored in temporary buffer **116** until instruction I2 completes. By having IEU **100** store each instruction's result in its preassigned place in the temporary buffers **116**, IEU **100** can execute instructions out of program order and still avoid the problems caused by output and anti-dependencies.

RRC **112** sends a bit map to an Instruction Scheduler **118** via a bus **120** indicating which instructions in window **102** are ready for issuing. Instruction decode logic (not shown) indicates to Issuer **118** the resource requirements for each instruction over a bus **123**. For each resource in IEU **100** (e.g., each functional unit being an adder, multiplier, shifter, or the like), Issuer **118** scans this information and selects the first and subsequent instructions for issuing by sending issue signals over bus **121**. The issue signals select a group of Register File Port MUXes (RPMs) **124** inside RRC **112** whose inputs are the addresses of each instruction's inputs.

Because the results may stay in temporary buffer **116** several cycles before going to register file **117**, a mechanism is provided to get results from temporary buffer **116** before they go to register file **117**, so the information can be used as operands for other instructions. This mechanism is called "result forwarding," and without it, Issuer **118** would not be able to issue instructions out of order. This result forwarding is done in register file **117** and is controlled by RRC **112**. The control signals necessary for performing the result forwarding will become evident to those skilled in the art, as should the random logic used for generating such control signals.

If an instruction is not dependent on any of the current instructions result forwarding is not necessary since the instruction's inputs are already in register file **117**. When Issuer **118** decides to execute that instruction, RRC **112** tells register file **117** to output its data.

RRC **112** contains three subsections: a Data Dependency Checker (DDC) **103**, Tag Assign Logic (TAL) **122** and Register File Port MUXes (RPM) **124**. DDC **108** determines where the input dependencies are between the current instructions. TAL **122** monitors the dependencies for Issuer **118** and controls result forwarding. RPM **124** is controlled by Issuer **118** and directs the outputs of TAL **122** to the appropriate register file address ports **119**. Instructions are passed to DDC **108** via bus **110**. All source registers are compared with all previous destination registers for each instruction in window **102**.

Each instruction has only one destination, which may be a double register in one embodiment. An instruction can only depend on a previous instruction and may have up to three source registers. There are various register file source and destination addresses that need to be checked against each other for any dependencies. As noted above, the eight bottom instructions corresponding to the lower two buckets are checked by DDC **108**. All source register addresses are compared with all previous destination register addresses for the instructions in window **102**.

For example, let's say a program has the following instruction sequence:

add R0, R1, R2 (0)

add R0, R2, R3 (1)

add R4, R5, R2 (2)

add R2, R3, R4 (3)

The first two registers in each instruction **0–3** are the source registers, and the last listed register in each instruction is the destination register. For example, **R0** and **R1** are the source registers for instruction **0** and **R2** is the destination register. Instruction **0** adds the contents of registers **0** and **1** and stores the result in **R2**. For instructions **1–3** in this example, the following are the comparisons needed to evaluate all of the dependencies:

I1S1, I1S2 vs. I0D

I2S1, I2S2 vs. I1D, I0D

I3S1, I3S2 vs. I2D, I1D, I0D

The key to the above is as follows: IXRS1 is the address of source (input) number **1** of instruction X; IXRS2 is the address of source (input) number **2** of instruction X; and IXD is the address of the destination (output) of instruction x Note also that RRC **112** can ignore the fact that instruction **2** is output dependent on instruction **0**, because the processor has a temporary buffer where instruction **2**'s result can be stored without interfering with instruction **0**'s result. As discussed before, instruction **2**'s result will not be moved from temporary buffers **116** to register file **117** until instructions **0** and 1's results are moved to register file **117**.

The number of instructions that can be checked by RRC **112** is easily scaleable. In order to check eight instructions at a time instead of four, the following additional comparisons would also need to be made:

I4S1, I4S2 vs I3D, I2D, I1D, I0D

I5S1, I5S2 vs I4D, I3D, I2D, I1D, I0D

I6S1, I6S2 vs I5D, I4D, I3D, I2D, I1D, I0D

I7S1, I7S2 vs I6D, I5D, I4D, I3D, I2D, I1D, I0D

There are several special cases that RRC **112** must handle in order to do the dependency check. First, there are some instructions that use the same register as an input and an output. Thus, RRC **112** must compare this source/destination register address with the destination register addresses of all previous instructions. So for instruction **7**, the following comparisons would be necessary:

I7S1,I7S2,I7S/D vs. I6D,I5D,I4D,I3D,I2D,I1D,I0D.

Another special case occurs when a program contains instructions that generate 64 bit outputs (called long-word operations). These instructions need two registers in which to store their results. In this embodiment, these registers must be sequential. Thus if RRC **112** is checking instruction **4**'s dependencies and instruction **1** is a long-word operation, then it must do the following comparisons:

I4S1,I4S2 vs. I3D,I2D,I1D,I1D+1,I0D

Sometimes, instructions do not have destination registers. Thus RRC **112** must ignore any dependencies between instructions without destination registers and any future instructions. Also, instructions may not have only one valid source register, so RRC **112** must ignore any dependencies between the unused source register (usually S2) and any previous instructions.

RRC **112** is also capable of dealing with multiple register files. When using multiple register files, dependencies only occur when one instruction's source register has the same address and is in the same register file as some other instruction's destination register. RRC **112** treats the infor-

mation regarding which register file a particular address is from as part of the address. For example, in an implementation using four 32 bit register files, RRC **112** would do 7 bit compares instead of 5 bit compares (5 for the address and 2 for the register file).

Signals indicating which instructions are long-word operations or have invalid source or destination registers are sent to RRC **112** from Instruction Decode Logic (IDL; not shown). IDL also tells RRC **112** which register file each instruction's sources and destinations will come from or go to.

A block diagram of DDC **108** is shown in FIG. **2**. Source address signals arrive from IFIFO **101** for all eight instructions of window **102**. Additional inputs include long-word load operation flags, register file decode signals, invalid destination register flags, destination address signals and addressing mode flags for all eight instructions.

DDC **208** comprises 28 data dependency blocks **204**. Each block **204** is described in a KEY **206**. Each block **204** receives 3 inputs, IXS1, IXS2 and IXS/D. IXS1 is the address of source (input) number **1** of instruction X; IXS2 is the address of source (input) number **2** of instruction X; and IXS/D is the address of the source/destination (input) of instruction X Each block **204** also receives input IYS/D, which is the destination register address for some previous instruction Y. A top row **208**, for example, receives I0S/D, which is the destination register address for instruction **0**. Each block **204** outputs the data dependency results to one of a corresponding bus line **114**. For example, the address of I2S/D must be checked with operand addresses S1, S2 and S/D of instructions **7**, **6**, **5**, **4**, and **3**.

Each block **204** performs the three comparisons. To illustrate these comparisons, consider a generic block **700** shown in FIG. **7**, which compares instruction Y's source/destination operand with each operand of instruction X In this example, the three following comparisons must be made:

IXS1=IYS/D

IXS2=IYS/D

IXS/D=IYS/D

These comparisons are represented by three comparator blocks **702**, **704** and **706**, respectively. One set of inputs to comparator blocks **702**, **704** and **706** are the bits of the IYS/D field, which is represented by number **708**. Comparator block **702** has as its second set of inputs the bits of the IXS1. Similarly, comparator block **704** has as its second set of inputs the bits of the IXS1, and comparator block **706** has as its second set of inputs the bits of the IXS/D.

In a preferred embodiment, the comparisons performed by blocks **702**, **704** and **706** can be performed by random logic. An example of random logic for comparator block **706** is shown in FIG. **8**. Instruction Y's source/destination bits [**6**:**0**] are shown input from the right at reference number **802** and instruction X's source/destination bits [**6**:**0**] are shown input from the top at reference number **804**. The most significant bit (MSB) is bit **6** and the least significant bit (LSB) is bit **0**. The corresponding bits from the two operands are fed to a set of seven exclusive NOR gates (XNORS) **806**. The outputs of XNORs **806** are then ANDed by a seven input AND gate **808**. If the corresponding bits are the same, the output of XNOR **806** will be logic high. When all bits are the same, all seven XNOR **806** outputs are logic high and the output of AND gate **808** is logic high, this indicates that there is a dependency between IXS/D and IYS/D.

The random logic for comparator blocks **702** and **704** will be identical to that shown in FIG. **8**. The present invention contemplates many other random logic circuits for performing data dependency checking, as will become evident to those skilled in the art without departing from the spirit of this example.

As will further become evident to those skilled in the art, various implementation specific special cases can arise which require additional random logic to perform data dependency checking. An illustrative special data dependency checking case is for long word handling.

As mentioned before, if a long word operation writes to register X the first 32 bits are written to register X and the second 32 bits are written to register X+1. The data dependency checker therefore needs to check both registers when doing a comparison. In a preferred embodiment, register X is an even register, X+1 is an odd register and thus they only differ by the LSB. The easiest way to check both registers at the same time is to simply ignore the LSB. In the case of a store long (STLG) or load long (LDLG) operation, if X and Y only differ by the LSB bit [**0**], the logic in FIG. **8** would cause there to be no dependency, when there really is a dependency. Therefore, for a long word operation the STLG and LDLG flags must be ORed with the output of the [**0**] bit XNOR to assure that all dependencies are detected.

A data dependency check flowchart for IXS1 and IYS/D is shown in FIG. **5**. DDC **108** first checks whether IXS1 and IYS/D are in the same register file, as shown at a conditional block **502**. If they are not in the same register file there is no dependency. This is shown at block a **504**. If there is a dependency, DDC **108** then determines whether IXS1 and IYS/D are in the same register, as shown at a block **506**. If they are not in the same register, flow proceeds to a conditional block **508** where DDC **108** determines whether IY is a long word operation. If IY is not a long word operation there is no dependency and flow proceeds to a block **504**. If IY is a long word operation, flow then proceeds to a conditional statement **510** where DDC **108** determines whether IXS1 and IYS/D+1 are the same register. If they are not, there is no dependency and flow proceeds to a block **504**. If IXS1 and IYS/D+1 are the same register, flow proceeds to a conditional block **512** where DDC **108** determines if IY has a valid destination. If it does not have a valid destination, there is no dependency and flow proceeds to block **504**. If IY does have a valid destination, flow proceeds to a conditional block **514** where DDC **108** determines if IXS1 has a valid source register. Again, if no valid source register is detected there is no dependency, and flow proceeds to a block **504**. If a valid source register is detected, DDC **108** has determined that there is a dependency between IXS1 and IYX/D, as shown at a block **516**.

A more detailed discussion of data dependency checking is found in commonly owned, copending application Ser. No. 07/860,718, the disclosure of which is incorporated herein by reference.

Because it is possible that an instruction might get one of its inputs from a register that was written to by several other instructions, the present invention must choose which one is the real dependency. For example, if instructions **2** and **5** write to register **4** and instruction **7** reads register **4**, then instruction **7** has two possible dependencies. In this case, it is assumed that since instruction **5** came after instruction **2** in the program, the programmer intended instruction **7** to use instruction **5**'s result and not instruction **2**'s. So, if an instruction can be dependent on several previous instructions, RRC **112** will consider it to be dependent on the highest numbered previous instruction.

Once TAL **122** has determined where the real dependencies are, it must locate the inputs for each instruction. In a preferred embodiment of the present invention, the inputs can come from the actual register file or an array temporary buffers **116**. RRC **112** assumes that if an instruction has no dependencies, its inputs are all in the register file. In this case, RRC **112** passes the IXS1, IXS2 and IXS/D addresses that came from IFIFO **102** to the register file. If an instruction has a dependency, then RRC **112** assumes that the data is in temporary buffers **116**. Since RRC **112** knows which previous instruction each instruction depends on, and since each instruction always writes to the same place in temporary buffers **116**, RRC **112** can determine where in temporary buffers **116** an instruction's inputs are stored. It sends these addresses to register file read ports **119** and register file **117** outputs the data from temporary buffers **116** so that the instruction can use it.

The following is an example of tag assignments:

0: add r0, r1, r2

1: add r0, r2, r3

2: add r4, r5, r2

3: add r2, r3, r4

The following are the dependencies for the above operations (dependencies are represented by the symbol "#"):

I1S2#I0S/D

I3S1#I0S/D

I3S1#I2S/D

I3S2#I1S/D

First, look at I0; since it has no dependencies, its tags are equal to its original source register addresses:

I0S1 TAG=I0S1=r0

I0S2 TAG=I0S2=r1

I0S/D TAG=I0S/D=r2

I1 has one dependency, and its tags are as follows:

I1S1 TAG=I1S1=r0

I1S2 TAG=I0S/D=t0

where: (t0=inst. 0's slot in temporary buffer)

I1S/D TAG=I1/ID=r3

I2 is also independent:

I2S1 TAG=I2S1=r4

I2S2 TAG=I2S2=r5

I2S/D TAG=I2S/D=r2

I3S1 has two possible dependencies, I0S/D and I2S/D. Because TAL **122** must pick the last one (highest numbered one), I2S/D is chosen.

I3S1 TAG=I2S/D=t2

I3S2 TAG=I1S/D=t1

I3S/D TAG=I3S/D=r4

These tags are then sent to RPM **124** via bus **126** to be selected by Issuer **118**. At the same time TAL **122** is preparing the tags, it is also monitoring the outputs of DCL **130** and passing them on to Issuer **118** using bus **120**. TAL **122** chooses the proper outputs of DCL's **130** to pass to Issuer **118** by the same method that it chooses the tags that it sends to RPM **124**.

Continuing the example, TAL **122** sends the following ready signals to Issuer **118**:

I0S1 INFO=1

(Inst **0** is independent so it can start immediately)

I0S2 INFO=1

I0S/D INFO=1

I1S1 INFO=1

I1S2 INF0=DONE[0]

(DONE[**0**]=1 when I0 is done)

I1S/D INFO=1

I2S1 INFO=1

I2S2 INFO=1

I2S/D INFO=1

I3S1 INFO=DONE[2]

I3S2 INFO=DONE[1]

I3S/D READ=1

(The DONE signals come from DCL **130** via a bus **132**. In connection with the present invention, the term "done" means the result of the instruction is in a temporary buffer or otherwise available at the output of a functional unit. Contrastingly, the term "terminate" means the result of the instruction is in the register file.)

Turning now to FIG. **3**, a representative block diagram of TAL **122** will be discussed. TAL **122** comprises **8** tag assignment logic blocks **302**. Each TAL block **302** receives the corresponding data dependency results via buses **114**, as well as further signals that come from the computer's Instruction Decode and control logic (not shown). The BKT bit signal forms the least significant bit of the tag. DONE[X] flags are for instructions **0** through **6**, and indicate if instruction X is done. DBLREG[X] flags indicates which, if any, of the instructions is a double (long) word. Each TAL block **302** also receives its own instructions register addresses as inputs. The Misc. signals, DBLREG and BKF signals are all implementation dependent control signals. Each TAL block **302** outputs 3 TAGs **126** labeled IXS1, IXS2 and IXS/D, which are 6 bits. TAL **122** outputs the least significant 5 bits of each TAG signal to RPMs **124** and the most significant TAG to Issuer **118**.

Each block **302** of FIG. **3** comprises three Priority Encoders (PE), one for **S1**, one for S2 and one for S/D. There is one exception however. **I10** requires no tag assignment. Its tags are the same as the original S1, S2 and S/D addresses, because I0 is always independent.

An illustrative PE is shown in FIG. 9. PE **902** has eight inputs **904** and eight outputs **906**. Inputs **904** for PE **902** are outputs **114** from DDC **108** which show where dependencies exist. For example, in the case of source register **1** (S1), I7S1 tag assign PE **902**'s seven inputs are the seven outputs **114**

of DDC **108** that indicate whether I7S1 is dependent on I6D, whether I7S1 is dependent on I5D, and so on down to whether I7S1 is dependent on I0D. An eighth input, shown at reference number **908**, is always tied high because there should always be an output from PE **902**.

As stated before, if an instruction depends on several previous instructions, PE **902** will select and output only the most previous instruction (in program order) on which there is a dependency. This is accomplished by connecting the signal showing if there is a dependency on the most previous instruction to the highest priority input of the PE **902** and the signal showing if there is a dependency on the second most previous instruction to the input of PE **902** with the second highest priority and so on for all previous instructions. The input of the PE **902** with the lowest priority is always tied high so that at least one of PE **902**'s outputs will be asserted.

Outputs **906** are used as select lines for a MUX **910**. MUX **910** has eight inputs **912** to which the tags for each instruction are applied.

To illustrate this, assume that I7 depends on I6 and I5, then, since I6 has a higher priority than I5, the bit corresponding to I6 at outputs **906** of PE **902** will be high. At the corresponding input **912** of MUX **910** will be I6's tag for S1 (recall PE **902** is for I7S1). Because I7 is dependent on I6, the location of I6's result must be output from MUX **910** so that it can be used by I7. I6's tag will therefore be selected and output on an output line **914**. I6's done flag, DONE[6] must also be output from MUX **910** so that Issuer **118** will know when I7's input is ready. This data is passed to Issuer **118** via bus **120**. Since an instruction can have up to three sources, TAL **122** monitors up to three dependencies for each instruction and sends three vectors for each instruction (totaling 24 vectors) to Issuer **118**. If an instruction is independent, TAL **122** signals to Issuer **118** that the instruction can begin immediately.

The MSB of the tag outputs which are sent to RPMs **124** is used to indicate if the address is a register file address or a temporary buffer address. If an instruction is independent, then the five LSB outputs indicate the source register address. For instructions that have dependencies: the second MSB indicates that the address is for a 64 bit valve; the third through fifth MSB outputs specie the temporary buffer address; and the LSB output indicates which bucket is the current bucket, which is equal to the BKT signal in TAL **122**.

Like DDC **108**, TAL **122** has numerous implementation dependent situations, (i.e., special cases) that it handles. First, in an embodiment of the present invention, register number **0** of the register file is always equal to 0. Therefore, even if one instruction writes to register **0** and another reads from register **0**, there will be no dependency between them. TAL **122** receives three signals from Instruction Decode Logic (IDL; not shown) for each instruction to indicate if one of that instruction's sources is register **0**. If any of those is asserted, TAL **122** will ignore any dependencies for that particular input of that instruction.

Another special case occurs because under some circumstances, an instruction in bucket **0** will be guaranteed to not have any of the instructions in bucket **1** dependent on it. A four bit signal called $BKT1_{13}$ $NODEP_{13}$ is sent to RRC **112** from the IEU control logic (not shown) and if $BKT1_{13}$ NODEP[X]=1 then RRC **112** knows to ignore any dependencies between instructions, 4,5,6 or 7 and instruction X An example for TAG assignment of instruction **7**'s source **1** (I7S1) is shown in a flowchart in FIGS. **6A–6B**. TAL **122** first determines whether I7S1 is register **0**, as shown at a conditional block **602**. If the first source operand for I7 is register **0**, the TAG is set equal to zero, and the I7S1's INFO flag is set equal to one, as shown in a block **604**. If the first source operand (S1) for I7 is not register **0**, TAL **122** then determines if I7S1 is dependent on I6S/D, as shown at a conditional block **606**. If I7S1 is dependent on I6S/D flow then proceeds to a block **610** where I7S1's TAG is set equal to {**1**,DBLREG[**6**],**0**,**1**,**0**,BKT} and I7S1's INFO flag is set equal to DONE[**6**], as shown at a block **610**. If either of the condition tested at a conditional block **606** is not met, flow proceeds to conditional block **612** where TAL **122** determines if I7S1 is dependent on I5S/D. If there is a dependency, flow then proceeds to block **616** where TAL **122** sets I7S1's TAG equal to {**1**,DBLREG[**5**],**0**,**0**,**1**,BKT} and I7S1's INFO flag is set equal to DONE[**5**]. If the condition tested at block **612** is not met, flow proceeds to a block **618** where TAL **122** determines if I7S1 is dependent on I4S/D.

As evident by inspection of the remaining sections of FIGS. **6A** and **6B**, similar TAG determinations are made depending on whether I7S1 is dependent on I4S/D, I3S/D, I2S/D, I1S/D and I0S/ID, as shown at sections **620**, **622**, **624**, **626** and **628**, respectively. Finally, if instruction **7** is independent of instruction **0** or if all instructions in bucket **1** are independent of instruction **0** (i.e., if $BKT1_{13}$ NODEP [**0**]=1), as tested at a conditional block **630**, the flow proceeds to block **632** where TAL **122** sets I7S1's TAG equal to {**0**,**I7S1**} and I7S1's INFO flag equal to 1. It should be noted for the above example that I7S1 TAG signals are forwarded directly the register file port MUXes of register file **117**. The I7S1 INFO signals are sent to Issuer **118** to tell it when I7's S1 input is ready.

A representative block diagram of Issuer **118** is shown in FIG. **10**. In a preferred embodiment, Issuer **118** has one scanner block **1002** for each resource (functional unit) that has to be allocated. In this example, Issuer **118** has scanner blocks FU1, FU2, FU3, FU4 through FUn. Requests for functional units are generated from instruction information by decoding logic (not shown) in a known manner, which are sent to scanners **1002** via bus **123**. Each scanner block **1002** scans from instruction **I0** to I7 and selects the first request for the corresponding functional unit to be serviced during that cycle.

In the case of multiple register files (integer, floating and/or boolean), Issuer **118** is capable of issuing instructions having operands stored in different register files. For example, an ADD instruction may have a first operand from the floating point register file and a second operand from the integer register file. Instructions with operands from different register files are typically given higher issue priority (i.e., they are issued first). This issuing technique conserves processor execution time and functional unit resources.

In a further embodiment in which IEU **100** may include two ALU's, ALU scanning becomes a bit more complicated. For speed reasons, one ALU scanner block scans from **I0** to I7, while the other scanner block scans from I7 to **I0**. This is how two ALU requests are selected. With this scheme it is possible that an ALU instruction in bucket **1** will get issued before an ALU instruction in bucket **0**, while increasing scanning efficiency.

Scanner outputs **1003** are selected by MUXing logic **1004**. A set of SELect inputs **1006** for MUX **1004** receive three 8-bit vectors (one for each operand) from TAL **122** via bus **120**. The vectors indicate which of the eight instructions have no dependencies and are ready to be issued. Issuer **118** must wait for this information before it can start to issue any instructions. Issuer **118** monitors these vectors and when all three go high for a particular instruction, Issuer **118** knows that the inputs for that instruction are ready. Once the

necessary functional unit is ready, the issuer can issue that instruction and send select signals to the register file port MUXes to pass the corresponding instructions outputs to register file **117**.

In a preferred embodiment of the present invention, after Issuer **118** is done it provides two **8**-bit vectors per register file back to RRC **112** via MUXOUTputs **1008** to bus **121**. These vectors indicate which instructions are issued this cycle, are used a select lines for RPMs **124**.

The maximum number of instructions that can be issued simultaneously for each register file is restricted by the number of register file read ports available. A data dependency with a previous uncompleted instruction may prevent an instruction from being issued. In addition, an instruction may be prevented from being issued if the necessary functional unit is allocated to another instruction.

Several instructions, such as load immediate instructions, Boolean operations and relative conditional branches, may be issued independently, because they may not require resources other than register file read ports or they may potentially have no dependencies.

The last section of RRC **112** is the register file port MUX (RPM) section **124**. The function of RPMs **124** is to provide a way for Issuer **118** to get data out of register files **117** for each instruction to use. RPMs **124** receive tag information via bus **126**, and the select lines for RPMs **124** come from Issuer **118** via a bus **121** and also from the computer's IEU control logic. The selected TAGs comprise read addresses that are sent to a predetermined set of ports **119** of register file **117** using bus **128**.

The number and design of RPMs **124** depend on the number of register files and the number of ports on each register file. One embodiment of RPMs **124** is shown in FIG. **4**. In this embodiment, RPMs **124** comprises **3** register port file MUXes **402**, **404** and **406**. MUX **402** receives as inputs the TAGs of instructions **0–7** corresponding to the source register field S1 that are generated by TAL **122**. MUX **404** receives as inputs the TAGs of instructions **0–7** corresponding to the source register field S2 that are generated by TAL **122**. MUX **406** receives as inputs the TAGs of instructions **0–7** corresponding to the source/destination register field S/D that are generated by TAL **122**. The outputs of MUXes **402**, **404** and **406** are connected to the read addresses ports of register file **117** via bus **128**.

RRC **112** and Issuer **118** allow the processor to execute instructions simultaneously and out of program order. An IEU for use with the present invention is disclosed in commonly owned, co-pending application Ser. No. 07/817,810, the disclosure of which is incorporated herein by reference.

While various embodiments of the present invention have been described above, it should be understood that they have been presented by way of example, and not limitation. Thus the breadth and scope of the present invention should not be limited by any of the above-described exemplary embodiments, but should be defined only in accordance with the following claims and their equivalents.

What is claimed is:

1. A system for register renaming in a computer system capable of out-of-order instruction execution, comprising:

a temporary buffer comprising a plurality of storage locations for storing execution results, wherein an execution result for an instruction in an instruction window is stored at one of said plurality of storage locations, said one of said plurality of storage locations being assigned to said instruction in said instruction window; and

tag assignment logic for receiving data dependency results from a data dependency checker and for outputting a tag comprising a temporary buffer storage location address in place of a register address for an operand of a first instruction, wherein said temporary buffer storage location address is an address of said operand in one of said plurality of storage locations if said first instruction is dependent on a previous one of said plurality of instructions in said instruction window for said operand.

2. The register renaming system of claim **1**, further comprising termination logic that transfers the execution results in said plurality of storage locations in said temporary buffer to register file locations in-order based on the order of instructions in said instruction window.

3. The register renaming system of claim **2**, wherein said termination logic transfers a plurality of execution results from said temporary buffer to said register file simultaneously.

4. The register renaming system of claim **3**, wherein said termination logic transfers an execution result for an instruction from said temporary buffer to said register file when all execution results for all prior instructions are retirable.

5. The register renaming system of claim **1**, wherein said tag further comprises a 1-bit identifier that indicates whether said address within said tag is an address within a register file or within said plurality of storage locations.

6. The register renaming system of claim **1**, further comprising register file port MUXes that pass said tags to read address ports of said temporary buffer for accessing said instruction execution results.

7. A computer system, comprising:

a memory unit for storing program instructions;

a bus coupled to said memory unit for retrieving said program instructions; and

a processor coupled to said bus, wherein said processor comprises a register renaming system, comprising:

a temporary buffer comprising a plurality of storage locations for storing execution results, wherein an execution result for an instruction in an instruction window is stored at one of said plurality of storage locations, said one of said plurality of storage locations being assigned to said instruction in said instruction window; and

tag assignment logic that receives data dependency results from a data dependency checker and outputs a temporary buffer storage location address in place of a register address for an operand of a first instruction if said first instruction is dependent on a previous one of said plurality of instructions in said instruction window for said operand, wherein said temporary buffer storage location address is an address of said operand in one of said plurality of storage locations.

8. The computer system of claim **7**, wherein said processor further comprises termination logic that transfers the execution results in said plurality of storage locations in said temporary buffer to register file locations in-order based on the order of instructions in said instruction window.

9. The computer system of claim **8**, wherein said termination logic transfers a plurality of execution results from said temporary buffer to said register file simultaneously.

10. The computer system of claim **9**, wherein said termination logic transfers an execution result for an instruction from said temporary buffer to said register file when all execution results for all prior instructions are retirable.

11. The computer system of claim **7**, wherein said tag comprises an address and a 1-bit identifier that indicates

whether said address within said tag is an address within a register file or said plurality of storage locations.

12. The computer system of claim 7, wherein said processor further comprises register file port MUXes that pass said tag to read address ports of said temporary buffer for accessing said execution results.

13. A register renaming method, comprising the steps of:

(1) storing, in a temporary buffer, out-of-order execution results in storage locations assigned to instructions in an instruction window;

(2) generating at least one tag to specify an address in said temporary buffer at which said out-of-order execution results are temporarily stored; and

(3) outputting one of said at least one tag comprising an address in place of a register address for an operand of a first instruction if a data dependency result indicates that said first instruction is dependent on a previous instruction in said instruction window, wherein said tag comprises an address of said operand in said temporary buffer.

14. The register renaming method of claim 13, further comprising the step of transferring said out-of-order execution results in said temporary buffer to a register file in-order based on the order of instructions in said instruction window.

15. The register renaming method of claim 14, further comprising the step of transferring a plurality of execution results from said temporary buffer to said register file simultaneously.

16. The register renaming method of claim 15, further comprising the step of transferring an out-of-order execution result from said temporary buffer to said register file when all execution results for all prior instructions are retirable.

17. The register renaming method of claim 13, further comprising the step of determining data dependencies between the instructions in said instruction window to produce said data dependency results.

18. The register renaming method of claim 13, wherein said step (2) further comprises the step of generating tags that comprise an address and a 1-bit identifier that indicates whether said address within said tags is an address within a register file or said temporary buffer.

19. The register renaming method of claim 13, further comprising the step of passing said tags to read address ports of said temporary buffer for accessing said out-of-order execution results.

* * * * *

EXHIBIT K

US005493687A

United States Patent [19]

Garg et al.

[11] Patent Number: 5,493,687

[45] Date of Patent: Feb. 20, 1996

[54] RISC MICROPROCESSOR ARCHITECTURE IMPLEMENTING MULTIPLE TYPED REGISTER SETS

[75] Inventors: **Sanjiv Garg**, Fremont; **Derek J. Lentz**, Los Gatos; **Le T. Nguyen**, Monte Sereno; **Sho L. Chen**, Saratoga, all of Calif.

[73] Assignee: **Seiko Epson Corporation**, Tokyo, Japan

[21] Appl. No.: **726,773**

[22] Filed: **Jul. 8, 1991**

[51] Int. Cl.[6] **G06F 9/34**

[52] U.S. Cl. **395/800**; 395/375; 364/232.23; 364/245.1; 364/247; 364/DIG. 1; 364/258; 364/259.1

[58] Field of Search 395/800, 375, 395/425

[56] References Cited

U.S. PATENT DOCUMENTS

4,212,076	7/1980	Conners	364/706
5,125,092	6/1992	Prener	395/725
5,201,056	4/1993	Daniel et al.	395/800
5,241,636	8/1993	Kohn	395/375

FOREIGN PATENT DOCUMENTS

0170284	2/1986	European Pat. Off. .
0213843	3/1987	European Pat. Off. .
0241909	10/1987	European Pat. Off. .
0454636	10/1991	European Pat. Off. .
2190521	11/1987	United Kingdom .

OTHER PUBLICATIONS

Ruby B. Lee, "Precision Architecture," *IEEE Computer*, pp. 78–91, Jan. 1989.

Daryl Odnert et al., "Architecture and Computer Enhancements for PA–RISC Workstations," *Proc. from IEEE Compcon*, San Francisco, CA, pp. 214–218, Feb. 1991.

Maejima et al., "A 16–bit Microprocessor with Multi–Register Bank Architecture", Proc. Fall Joint Computer Conference, Nov. 2–6, 1986, pp. 1014–1019.

Groves et al., "An IBM Second Generation RISC Processor Architecture", 35th IEEE Computer Society International Conference, Feb. 26, 1990, pp. 166–172.

Miller et al., "Exploiting Large Register Sets", Microprocessors and Microsystems, vol. 14, No. 6, Jul. 1990, pp. 333–340.

Adams et al., "Utilising Low Level Parallelism in General Purpose Code: The HARP Project", Microprocessing and Microprogramming, vol. 29, No. 3, Oct. 1990, pp. 137–149.

Molnar et al. "Floating–Point Processor" 1989 IEEE.

Stevens et al; "HARP: A Parallel Pipelined RISC Processor"; Nov. 1989.

Birman et al; "Design of a High–Speed Arithmetic Datapath"; 1988.

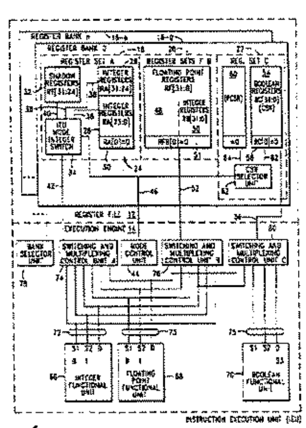
Paterson et al. "A VLSI RISC" *Computer* Sep. 1982.

Primary Examiner—Alyssa H. Bowler
Assistant Examiner—L. Donaghue
Attorney, Agent, or Firm—Sterne, Kessler, Goldstein & Fox

[57] ABSTRACT

A register system for a data processor which operates in a plurality of modes. The register system provides multiple, identical banks of register sets, the data processor controlling access such that instructions and processes need not specify any given bank. An integer register set includes first (RA[23:0]) and second (RA[31:24]) subsets, and a shadow subset (RT[31:24]). While the data processor is in a first mode, instructions access the first and second subsets. While the data processor is in a second mode, instructions may access the first subset, but any attempts to access the second subset are re-routed to the shadow subset instead, transparently to the instructions, allowing system routines to seemingly use the second subset without having to save and restore data which user routines have written to the second subset. A re-typable register set provides integer width data and floating point width data in response to integer instructions and floating point instructions, respectively. Boolean comparison instructions specify particular integer or floating point registers for source data to be compared, and specify a particular Boolean register for the result, so there are no dedicated, fixed-location status flags. Boolean combinational instructions combine specified Boolean registers, for performing complex Boolean comparisons without intervening conditional branch instructions, to minimize pipeline disruption.

5 Claims, 9 Drawing Sheets



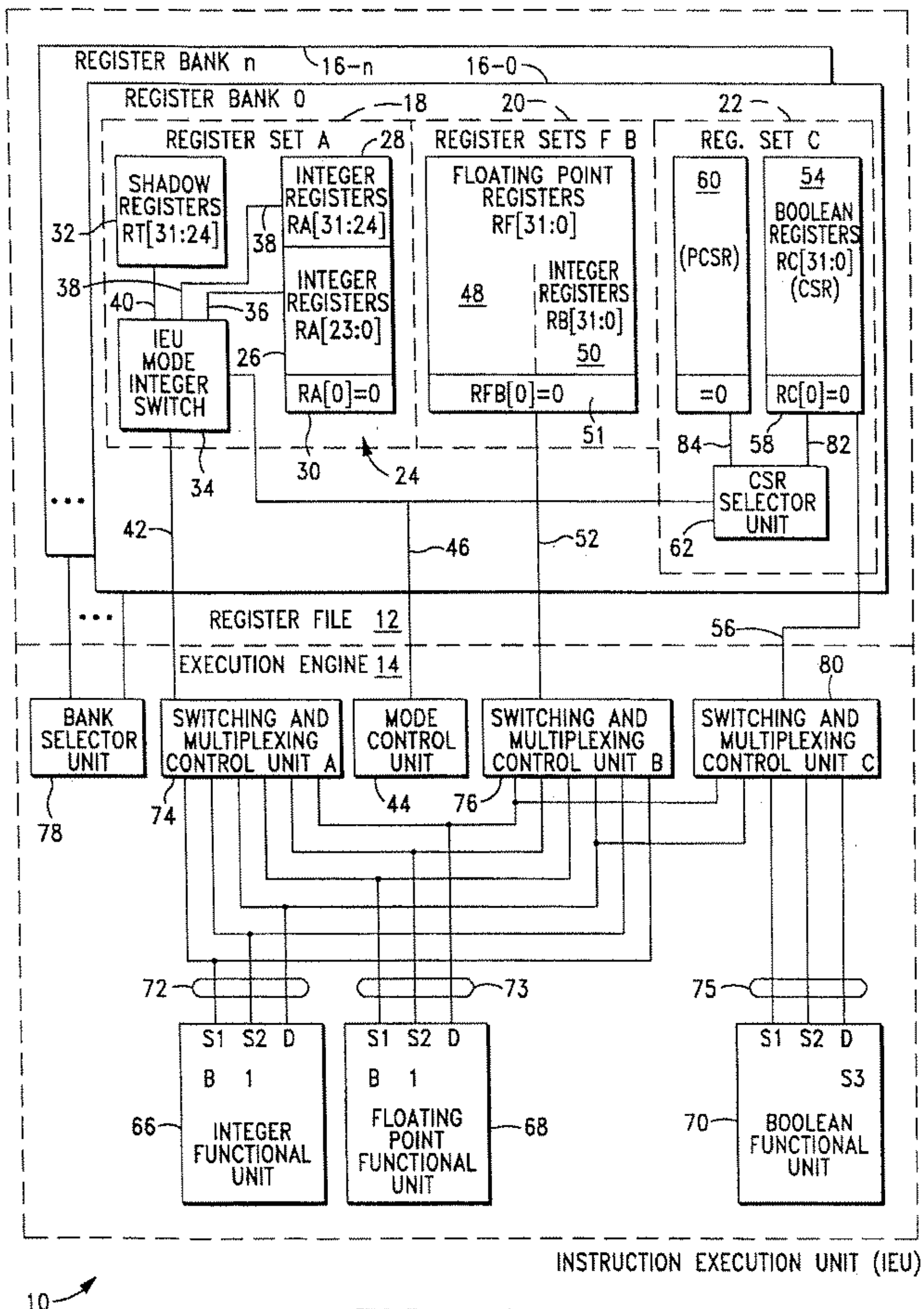
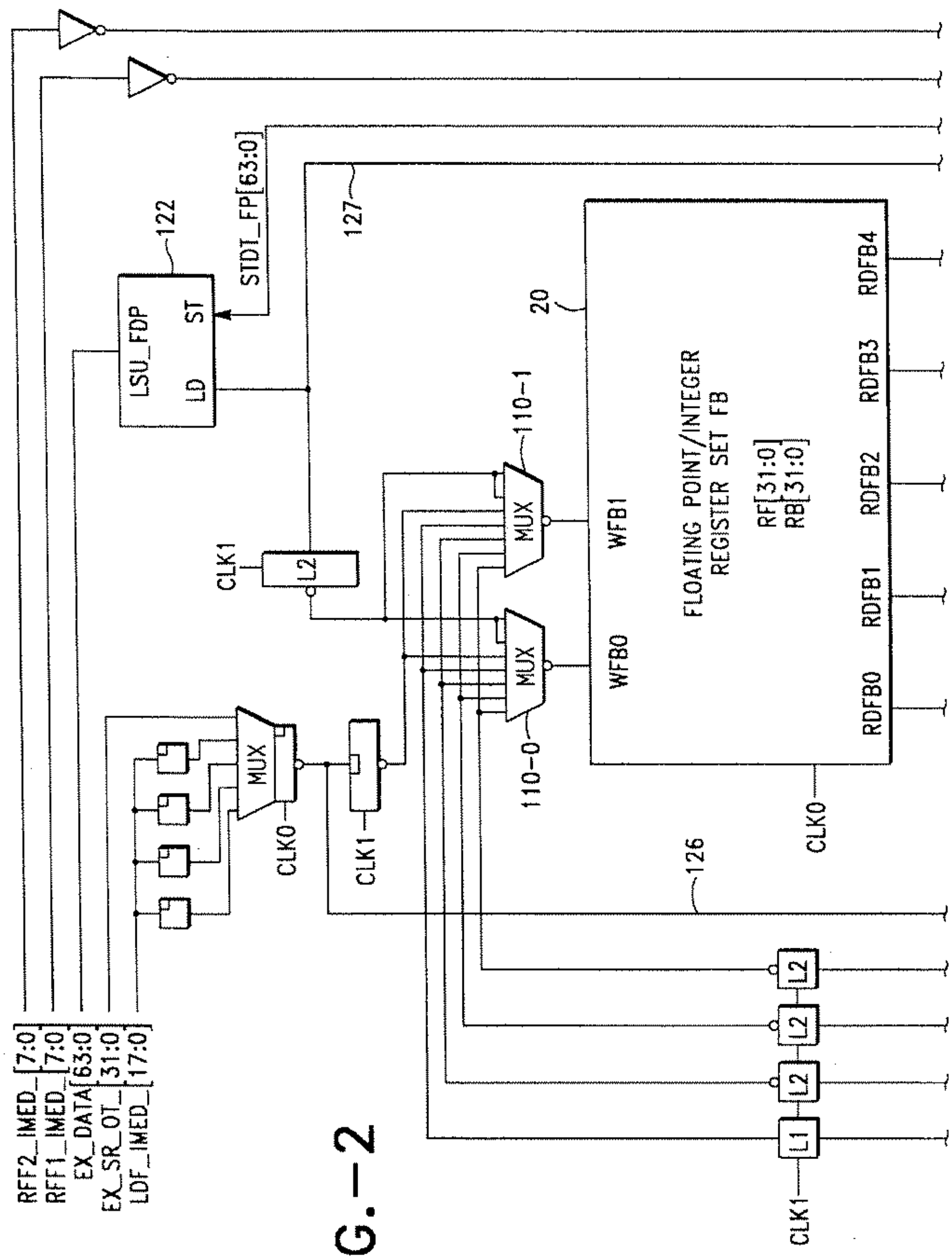

REGISTER BANK n
16-n
16-0
REGISTER BANK 0
18
20
22
REGISTER SET A
28
REGISTER SETS F B
REG. SET C
SHADOW REGISTERS RT[31:24]
INTEGER REGISTERS RA[31:24]
FLOATING POINT REGISTERS RF[31:0]
60
54
BOOLEAN REGISTERS RC[31:0] (CSR)
(PCSR)
32
38
38
40
36
IEU MODE INTEGER SWITCH
INTEGER REGISTERS RA[23:0]
48
INTEGER REGISTERS RB[31:0]
50
26
RA[0]=0
RFB[0]=0
=0
RC[0]=0
51
84
58
82
34
30
24
CSR SELECTOR UNIT
42
46
52
62
REGISTER FILE 12
56
EXECUTION ENGINE 14
80
BANK SELECTOR UNIT
SWITCHING AND MULTIPLEXING CONTROL UNIT A
MODE CONTROL UNIT
SWITCHING AND MULTIPLEXING CONTROL UNIT B
SWITCHING AND MULTIPLEXING CONTROL UNIT C
78
74
44
76
72
73
75
S1 S2 D
B 1
INTEGER FUNCTIONAL UNIT
S1 S2 D
B 1
FLOATING POINT FUNCTIONAL UNIT
S1 S2 D
S3
BOOLEAN FUNCTIONAL UNIT
66
68
70
INSTRUCTION EXECUTION UNIT (IEU)
10

FIG.—1



RFF2_IMED_[7:0]
RFF1_IMED_[7:0]
EX_DATA[63:0]
EX_SR_OT_[31:0]
LDF_IMED_[17:0]
LSU_FDP
LD
ST
122
STDT_FP[63:0]
127
MUX
CLK0
CLK1
L2
110-1
110-0
20
WFB1
WFB0
FLOATING POINT/INTEGER
REGISTER SET FB
RF[31:0]
RB[31:0]
RDFB4
RDFB3
RDFB2
RDFB1
RDFB0
126
L1

FIG.—2

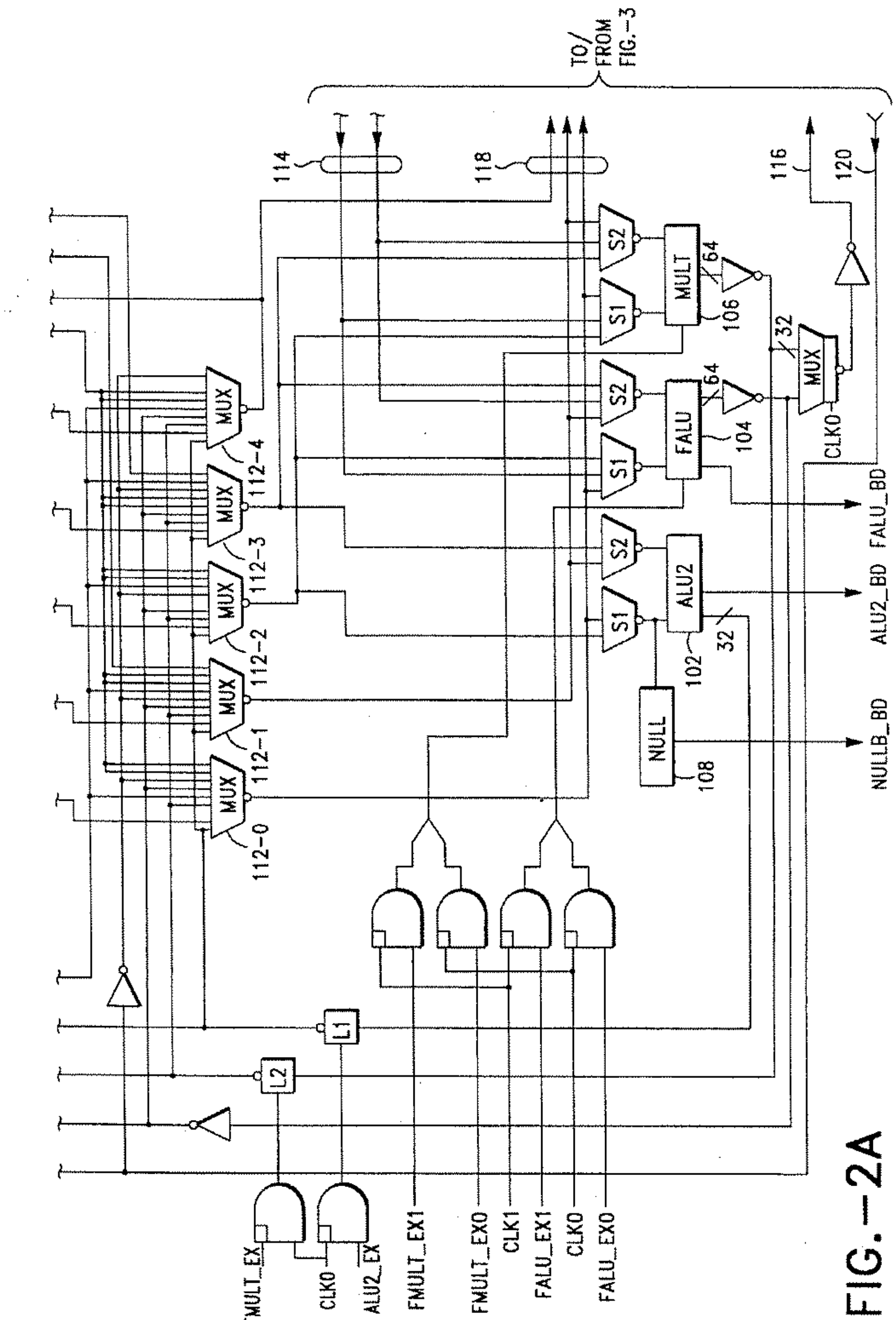

TO/
FROM
FIG.–3
114
118
116
120
MULT
FALU
ALU2
NULL
MUX
S1
S2
64
106
104
102
108
32
112-0
112-1
112-2
112-3
112-4
CLK0
FALU_BD
ALU2_BD
NULLB_BD
L1
L2
FMULT_EX
CLK0
ALU2_EX
FMULT_EX1
FMULT_EX0
CLK1
FALU_EX1
CLK0
FALU_EX0

FIG.–2A

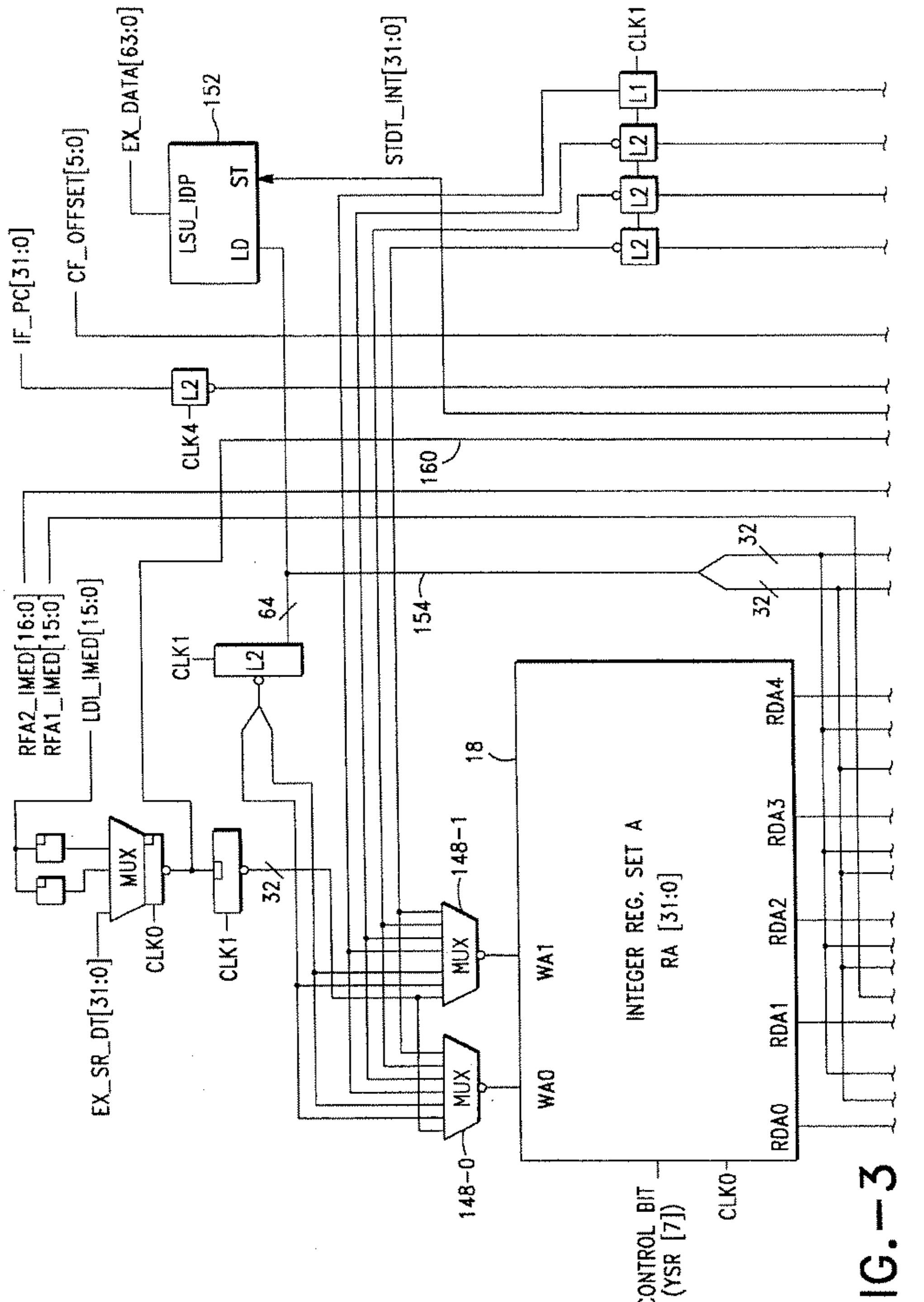

EX_DATA[63:0]
152
LSU_IDP
ST
LD
STDT_INT[31:0]
CLK1
L1
L2
L2
L2
IF_PC[31:0]
CF_OFFSET[5:0]
L2
CLK4
160
RFA2_IMED[16:0]
RFA1_IMED[15:0]
LDI_IMED[15:0]
64
CLK1
L2
154
32
32
18
MUX
CLK0
CLK1
32
EX_SR_DT[31:0]
148-1
MUX
148-0
MUX
WA1
WA0
INTEGER REG. SET A
RA [31:0]
RDA4
RDA3
RDA2
RDA1
RDA0
CONTROL BIT
(YSR [7])
CLK0

FIG.—3

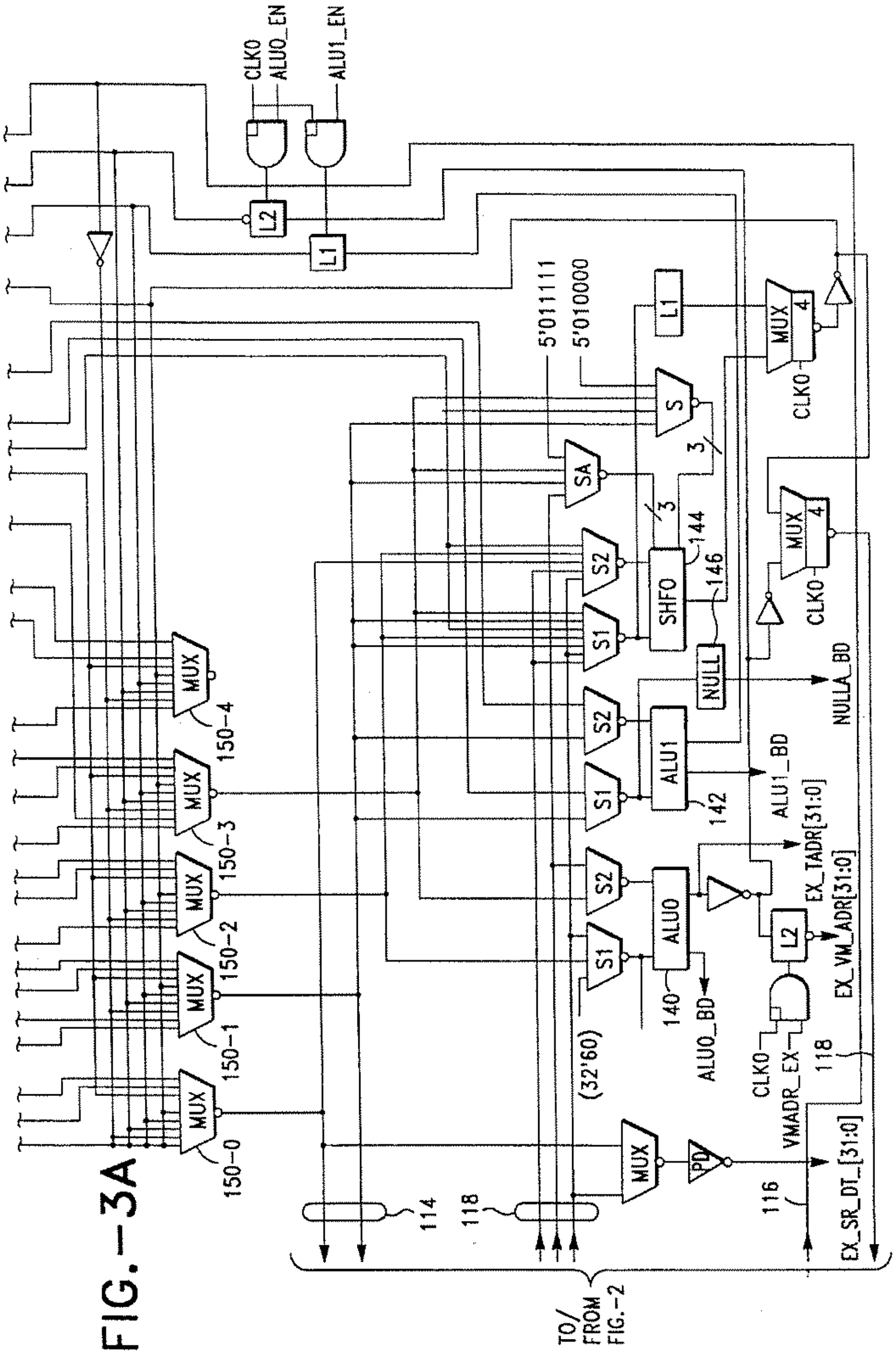

CLK0
ALU0_EN
ALU1_EN
L2
L1
5'011111
5'010000
L1
MUX
4
S
SA
3
3
CLK0
MUX
4
144
146
S2
S1
SHF0
CLK0
NULL
NULLA_BD
S2
S1
ALU1
ALU1_BD
142
MUX
150-4
MUX
150-3
MUX
150-2
MUX
150-1
MUX
150-0
S2
S1
ALU0
EX_TADR[31:0]
L2
EX_VM_ADR[31:0]
140
ALU0_BD
(32'60)
CLK0
VMADR_EX
118
MUX
PD
116
EX_SR_DT_[31:0]
114
118
TO/
FROM
FIG.-2

FIG.—3A

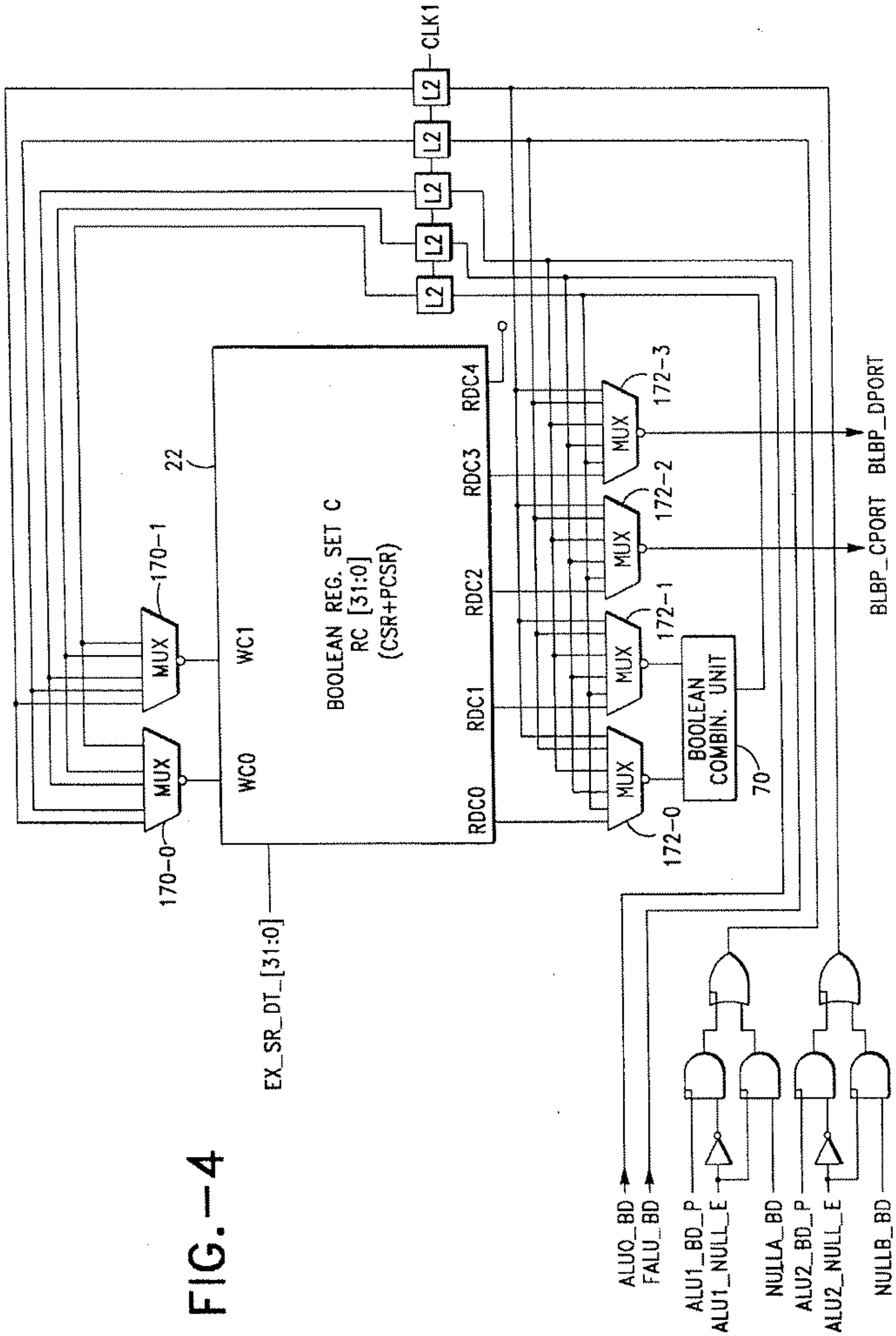

CLK1
L2
BOOLEAN REG. SET C
RC [31:0]
(CSR+PCSR)
22
WC1
WC0
RDC0
RDC1
RDC2
RDC3
RDC4
MUX
170-0
170-1
172-0
172-1
172-2
172-3
BOOLEAN COMBIN. UNIT
70
BLBP_CPORT
BLBP_DPORT
EX_SR_DT_[31:0]
ALU0_BD
FALU_BD
ALU1_BD_P
ALU1_NULL_E
NULLA_BD
ALU2_BD_P
ALU2_NULL_E
NULLB_BD

FIG.—4

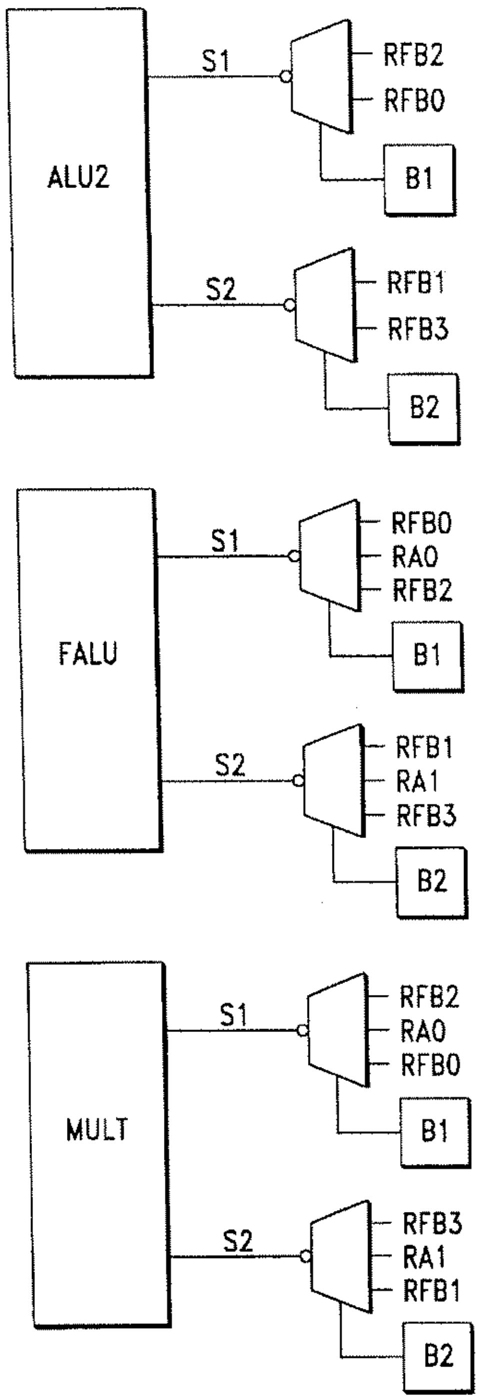

ALU2
S1
RFB2
RFB0
B1
S2
RFB1
RFB3
B2
FALU
S1
RFB0
RA0
RFB2
B1
S2
RFB1
RA1
RFB3
B2
MULT
S1
RFB2
RA0
RFB0
B1
S2
RFB3
RA1
RFB1
B2

FIG.—5

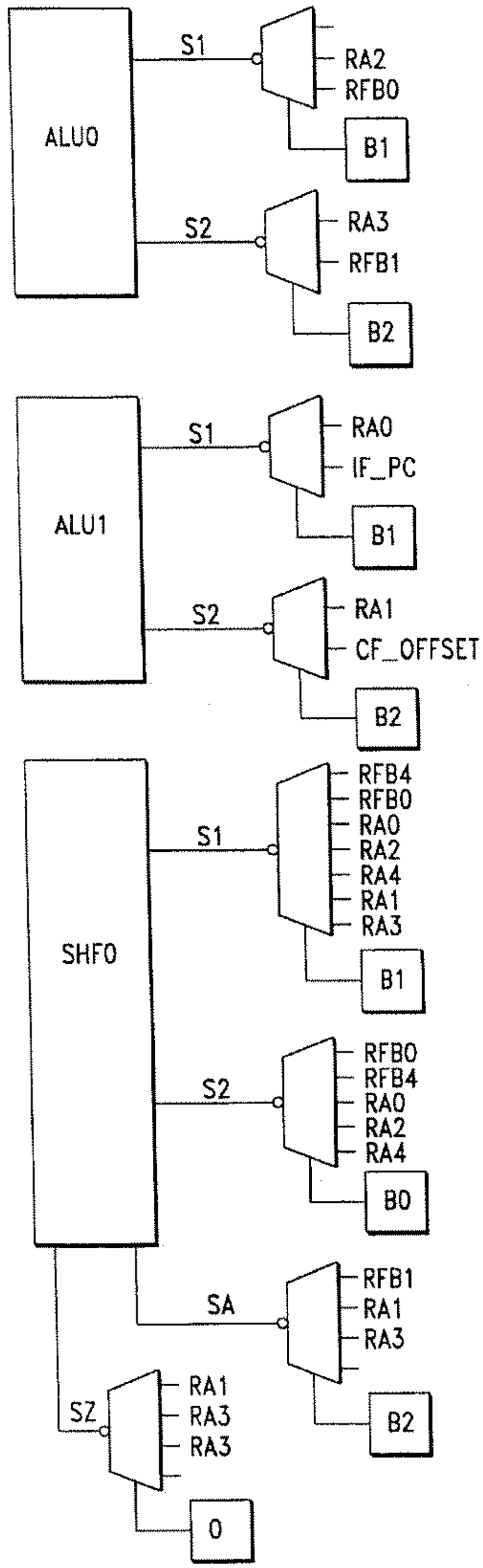

ALU0
S1
RA2
RFB0
B1
S2
RA3
RFB1
B2
ALU1
S1
RA0
IF_PC
B1
S2
RA1
CF_OFFSET
B2
SHF0
S1
RFB4
RFB0
RA0
RA2
RA4
RA1
RA3
B1
S2
RFB0
RFB4
RA0
RA2
RA4
B0
SA
RFB1
RA1
RA3
B2
SZ
RA1
RA3
RA3
0

FIG.—6

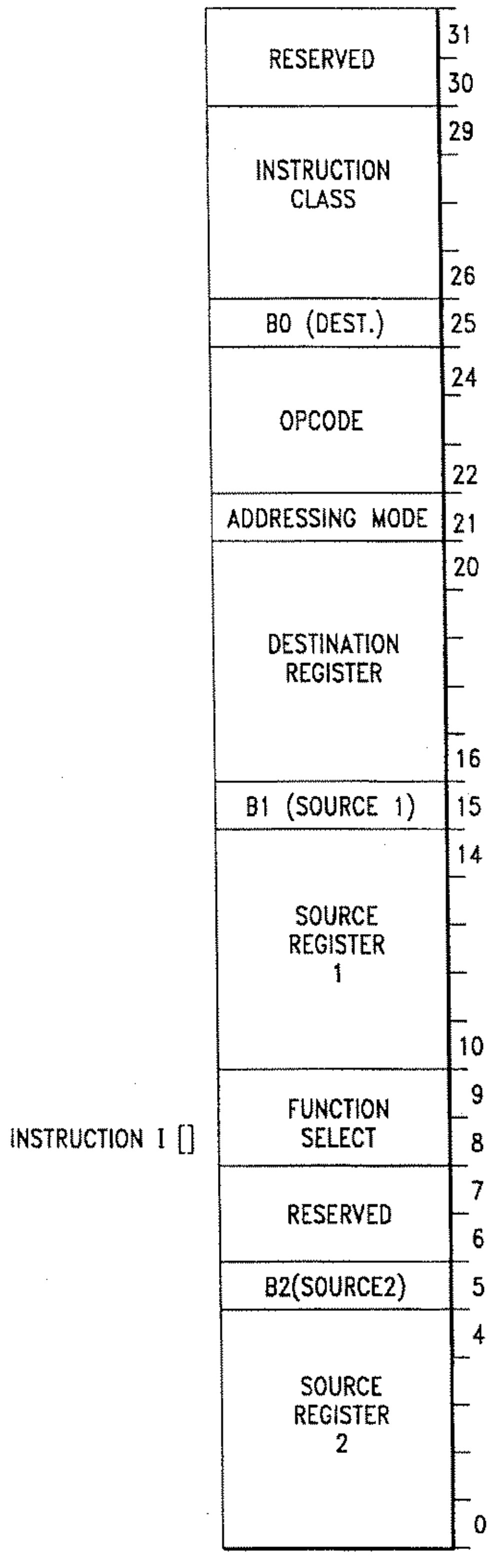

RESERVED
31
30
INSTRUCTION CLASS
29
26
B0 (DEST.)
25
OPCODE
24
22
ADDRESSING MODE
21
DESTINATION REGISTER
20
16
B1 (SOURCE 1)
15
SOURCE REGISTER 1
14
10
FUNCTION SELECT
9
8
RESERVED
7
6
B2(SOURCE2)
5
SOURCE REGISTER 2
4
0
INSTRUCTION I []

FIG.—7

RISC MICROPROCESSOR ARCHITECTURE IMPLEMENTING MULTIPLE TYPED REGISTER SETS

CROSS-REFERENCE TO RELATED APPLICATIONS

Applications of particular interest to the present application, include:

1. HIGH-PERFORMANCE RISC MICROPROCESSOR ARCHITECTURE, SC/Ser. No. 07/817,810, filed Jan. 8, 1992, which is a continuation of Ser. No. 07/727,066, filed Jul. 8, 1991, by Le Nguyen, et al.;

2. EXTENSIBLE RISC MICROPROCESSOR ARCHITECTURE, SC/Ser. No. 07/817,809, filed Jan. 8, 1992, which is a continuation of Ser. No. 07/727,058, filed Jul. 8, 1991, by Quang Trang, et al.

3. RISC MICROPROCESSOR ARCHITECTURE WITH ISOLATED ARCHITECTURAL DEPENDENCIES, SC/Ser. No. 07/817,807, filed Jan. 8, 1992, which is a continuation of Ser. No. 07/726,744, filed Jul. 8, 1991, by Yoshi Miyayama;

4. RISC MICROPROCESSOR ARCHITECTURE IMPLEMENTING FAST TRAP AND EXCEPTION STATE, Ser. No. 08/345,333 which is now U.S. Pat. No. 5,481,685, filed Nov. 21, 1994, which is a continuation of Ser. No. 07/171,968, filed Dec. 23, 1993, which is a continuation of Ser. No. 07/817,811, filed Jan. 8, 1992, which is a continuation of Ser. No. 07/726,942, filed Jul. 8, 1991 by Quang Trang;

5. SINGLE CHIP PAGE PRINTER CONTROLLER, SC/Ser. No. 07/817,813, filed Jan. 8, 1992, which is a continuation of Ser. No. 07/726,929, filed Jul. 8, 1991 by Derek Lentz; and

6. MICROPROCESSOR ARCHITECTURE CAPABLE OF SUPPORTING MULTIPLE HETEROGENEOUS PROCESSORS, SC/Ser. No. 07/726,893, filed Jul. 8, 1991, by Derek Lentz.

The above-identified Applications are hereby incorporated herein by reference, their collective teachings being part of the present disclosure.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates generally to microprocessors, and more specifically to a RISC microprocessor having plural, symmetrical sets of registers.

2. Description of the Background

In addition to the usual complement of main memory storage and secondary permanent storage, a microprocessor-based computer system typically also includes one or more general purpose data registers, one or more address registers, and one or more status flags. Previous systems have included integer registers for holding integer data and floating point registers for holding floating point data. Typically, the status flags are used for indicating certain conditions resulting from the most recently executed operation. There generally are status flags for indicating whether, in the previous operation: a carry occurred, a negative number resulted, and/or a zero resulted.

These flags prove useful in determining the outcome of conditional branching within the flow of program control. For example, if it is desired to compare a first number to a

second number and upon the conditions that the two are equal, to branch to a given subroutine, the microprocessor may compare the two numbers by subtracting one from the other, and setting or clearing the appropriate condition flags. The numerical value of the result of the subtraction need not be stored. A conditional branch instruction may then be executed, conditioned upon the status of the zero flag. While being simple to implement, this scheme lacks flexibility and power. Once the comparison has been performed, no further numerical or other operations may be performed before the conditional branch upon the appropriate flag; otherwise, the intervening instructions will overwrite the condition flag values resulting from the comparison, likely causing erroneous branching. The scheme is further complicated by the fact that it may be desirable to form greatly complex tests for branching, rather than the simple equality example given above.

For example, assume that the program should branch to the subroutine only upon the condition that a first number is greater than a second number, and a third number is less than a fourth number, and a fifth number is equal to a sixth number. It would be necessary for previous microprocessors to perform a lengthy series of comparisons heavily interspersed with conditional branches. A particularly undesirable feature of this serial scheme of comparing and branching is observed in any microprocessor having an instruction pipeline.

In a pipelined microprocessor, more than one instruction is being executed at any given time, with the plural instructions being in different stages of execution at any given moment. This provides for vastly improved throughput. A typical pipeline microprocessor may include pipeline stages for: (a) fetching an instruction, (b) decoding the instruction, (c) obtaining the instruction's operands, (d) executing the instruction, and (e) storing the results. The problem arises when a conditional branch instruction is fetched. It may be the case that the conditional branch's condition cannot yet be tested, as the operands may not yet be calculated, if they are to result from operations which are yet in the pipeline. This results in a "pipeline stall" which dramatically slows down the processor.

Another shortcoming of previous microprocessor-based systems is that they have included only a single set of registers of any given data type. In previous architectures, when an increased number of registers has been desired within a given data type, the solution has been simply to increase the size of the single set of those type of registers. This may result in addressing problems, access conflict problems, and symmetry problems.

On a similar note, previous architectures have restricted each given register set to one respective numerical data type. Various prior systems have allowed general purpose registers to hold either numerical data or address "data", but the present application will not use the term "data" to include addresses. What is intended may be best understood with reference to two prior systems. The Intel 8085 microprocessor includes a register pair "HL" which can be used to hold either two bytes of numerical data or one two-byte address. The present application's improvement is not directed to that issue. More on point, the Intel 80486 microprocessor includes a set of general purpose integer data registers and a set of floating point registers, with each set being limited to its respective data type, at least for purposes of direct register usage by arithmetic and logic units.

This proves wasteful of the microprocessor's resources, such as the available silicon area, when the microprocessor

is performing operations which do not involve both data types. For example, user applications frequently involve exclusively integer operations, and perform no floating point operations whatsoever. When such a user application is run on a previous microprocessor which includes floating point registers (such as the 80486), those floating point registers remain idle during the entire execution.

Another problem with previous microprocessor register set architecture is observed in context switching or state switching between a user application and a higher access privilege level entity such as the operating system kernel. When control within the microprocessor switches context, mode, or state, the operating system kernel or other entity to which control is passed typically does not operate on the same data which the user application has been operating on. Thus, the data registers typically hold data values which are not useful to the new control entity but which must be maintained until the user application is resumed. The kernel must generally have registers for its own use, but typically has no way of knowing which registers are presently in use by the user application. In order to make space for its own data, the kernel must swap out or otherwise store the contents of a predetermined subset of the registers. This results in considerable loss of processing time to overhead, especially if the kernel makes repeated, short-duration assertions of control.

On a related note, in prior microprocessors, when it is required that a "grand scale" context switch be made, it has been necessary for the microprocessor to expend even greater amounts of processing resources, including a generally large number of processing cycles, to save all data and state information before making the switch. When context is switched back, the same performance penalty has previously been paid, to restore the system to its former state. For example, if a microprocessor is executing two user applications, each of which requires the full complement of registers of each data type, and each of which may be in various stages of condition code setting operations or numerical calculations, each switch from one user application to the other necessarily involves swapping or otherwise saving the contents of every data register and state flag in the system. This obviously involves a great deal of operational overhead, resulting in significant performance degradation, particularly if the main or the secondary storage to which the registers must be saved is significantly slower than the microprocessor itself.

Therefore, we have discovered that it is desirable to have an improved microprocessor architecture which allows the various component conditions of a complex condition to be calculated without any intervening conditional branches. We have further discovered that it is desirable that the plural simple conditions be calculable in parallel, to improve throughput of the microprocessor.

We have also discovered that it is desirable to have an architecture which allows multiple register sets within a given data type.

Additionally, we have discovered it to be desirable for a microprocessor's floating point registers to be usable as integer registers, in case the available integer registers are inadequate to optimally to hold the necessary amount of integer data. Notably, we have discovered that it is desirable that such re-typing be completely transparent to the user application.

We have discovered it to be highly desirable to have a microprocessor which provides a dedicated subset of registers which are reserved for use by the kernel in lieu of at least a subset of the user registers, and that this new set of registers should be addressable in exactly the same manner as the register subset which they replace, in order that the kernel may use the same register addressing scheme as user applications. We have further observed that it is desirable that the switch between the two subsets of registers require no microprocessor overhead cycles, in order to maximally utilize the microprocessor's resources.

Also, we have discovered it to be desirable to have a microprocessor architecture which allows for a "grand scale" context switch to be performed with minimal overhead. In this vein, we have discovered that is desirable to have an architecture which allows for plural banks of register sets of each type, such that two or more user applications may be operating in a multi-tasking environment, or other "simultaneous" mode, with each user application having sole access to at least a full bank of registers. It is our discovery that the register addressing scheme should, desirably, not differ between user applications, nor between register banks, to maximize simplicity of the user applications, and that the system should provide hardware support for switching between the register banks so that the user applications need not be aware of which register bank which they are presently using or even of the existence of other register banks or of other user applications.

These and other advantages of our invention will be appreciated with reference to the following description of our invention, the accompanying drawings, and the claims.

SUMMARY OF THE INVENTION

The present invention provides a register file system comprising: an integer register set including first and second subsets of integer registers, and a shadow subset; a retypable set of registers which are individually usable as integer registers or as floating point registers; and a set of individually addressable Boolean registers.

The present invention includes integer and floating point functional units which execute integer instructions accessing the integer register set, and which operate in a plurality of modes. In any mode, instructions are granted ordinary access to the first subset of integer registers. In a first mode, instructions are also granted ordinary access to the second subset. However, in a second mode, instructions attempting to access the second subset are instead granted access to the shadow subset, in a manner which is transparent to the instructions. Thus, routines may be written without regard to which mode they will operate in, and system routines (which operate in the second mode) can have at least the second subset seemingly at their disposal, without having to expend the otherwise-required overhead of saving the second subset's contents (which may be in use by user processes operating in the first mode).

The invention further includes a plurality of integer register sets, which are individually addressable as specified by fields in instructions. The register sets include read ports and write ports which are accessed by multiplexers, wherein the multiplexers are controlled by contents of the register set-specifying fields in the instructions.

One of the integer register sets is also usable as a floating point register set. In one embodiment, this set is sixty-four bits wide to hold double-precision floating point data, but only the low order thirty-two bits are used by integer instructions.

The invention includes functional units for performing Boolean operations, and further includes a Boolean register

set for holding results of the Boolean operations such that no dedicated, fixed-location status flags are required. The integer and floating point functional units execute numerical comparison instructions, which specify individual ones of the Boolean registers to hold results of the comparisons. A Boolean functional unit executes Boolean combinational instructions whose sources and destination are specified registers in the Boolean register set. Thus, the present invention may perform conditional branches upon a single result of a complex Boolean function without intervening conditional branch instructions between the fundamental parts of the complex Boolean function, minimizing pipeline disruption in the data processor.

Finally, there are multiple, identical register banks in the system, each bank including the above-described register sets. A bank may be allocated to a given process or routine, such that the instructions within the routine need not specify upon which bank they operate.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of the instruction execution unit of the microprocessor of the present invention, showing the elements of the register file.

FIGS. 2, 3 and 4 are simplified schematic and block diagrams of the floating point, integer and Boolean portions of the instruction execution unit of FIG. 1, respectively.

FIGS. 5–6 are more detailed views of the floating point and integer portions, respectively, showing the means for selecting between register sets.

FIG. 7 illustrates the fields of an exemplary microprocessor instruction word executable by the instruction execution unit of FIG. 1.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

I. REGISTER FILE

FIG. 1 illustrates the basic components of the instruction execution unit (IEU) 10 of the RISC (reduced instruction set computing) processor of the present invention. The IEU 10 includes a register file 12 and an execution engine 14. The register file 12 includes one or more register banks 16-0 to 16-n. It will be understood that the structure of each register bank 16 is identical to all of the other register banks 16. Therefore, the present application will describe only register bank 16-0. The register bank includes a register set A 18, a register set FB 20, and a register set C 22.

In general, the invention may be characterized as a RISC microprocessor having a register file optimally configured for use in the execution of RISC instructions, as opposed to conventional register files which are sufficient for use in the execution of CISC (complex instruction set computing) instructions by CISC processors. By having a specially adapted register file, the execution engine of the microprocessor's IEU achieves greatly improved performance, both in terms of resource utilization and in terms of raw throughput. The general concept is to tune a register set to a RISC instruction, while the specific implementation may involve any of the register sets in the architecture.

A. Register Set A

Register set A 18 includes integer registers 24 (RA[31:0]), each of which is adapted to hold an integer value datum. In one embodiment, each integer may be thirty-two bits wide.

The RA[] integer registers 24 include a first plurality 26 of integer registers (RA[23:0]) and a second plurality 28 of integer registers (RA[31:24]). The RA[] integer registers 24 are each of identical structure, and are each addressable in the same manner, albeit with a unique address within the integer register set 24. For example, a first integer register 30 (RA[0]) is addressable at a zero offset within the integer register set 24.

RA[0] always contains the value zero. It has been observed that user applications and other programs use the constant value zero more than any other constant value. It is, therefore, desirable to have a zero readily available at all times, for clearing, comparing, and other purposes. Another advantage of having a constant, hard-wired value in a given register, regardless of the particular value, is that the given register may be used as the destination of any instruction whose results need not be saved.

Also, this means that the fixed register will never be the cause of a data dependency delay. A data dependency exists when a "slave" instruction requires, for one or more of its operands, the result of a "master" instruction. In a pipelined processor, this may cause pipeline stalls. For example, the master instruction, although occurring earlier in the code sequence than the slave instruction, may take considerably longer to execute. It will be readily appreciated that if a slave "increment and store" instruction operates on the result data of a master "quadruple-word integer divide" instruction, the slave instruction will be fetched, decoded, and awaiting execution many clock cycles before the master instruction has finished execution. However, in certain instances, the numerical result of a master instruction is not needed, and the master instruction is executed for some other purpose only, such as to set condition code flags. If the master instruction's destination is RA[0], the numerical results will be effectively discarded. The data dependency checker (not shown) of the IEU 10 will not cause the slave instruction to be delayed, as the ultimate result of the master instruction —zero— is already known.

The integer register set A 24 also includes a set of shadow registers 32 (RT[31:24]). Each shadow register can hold an integer value, and is, in one embodiment, also thirty-two bits wide. Each shadow register is addressable as an offset in the same manner in which each integer register is addressable.

Finally, the register set A includes an IEU mode integer switch 34. The switch 34, like other such elements, need not have a physical embodiment as a switch, so long as the corresponding logical functionality is provided within the register sets. The IEU mode integer switch 34 is coupled to the first subset 26 of integer registers on line 36, to the second subset of integer registers 28 on line 38, and to the shadow registers 32 on line 40. All accesses to the register set A 18 are made through the IEU mode integer switch 34 on line 42. Any access request to read or write a register in the first subset RA[23:0] is passed automatically through the IEU mode integer switch 34. However, accesses to an integer register with an offset outside the first subset RA[23:0] will be directed either to the second subset RA[31:24] or the shadow registers RT[31;24], depending upon the operational mode of the execution engine 14.

The IEU mode integer switch 34 is responsive to a mode control unit 44 in the execution engine 14. The mode control unit 44 provides pertinent state or mode information about the IEU 10 to the IEU mode integer switch 34 on line 46. When the execution engine performs a context switch such as a transfer to kernel mode, the mode control unit 44 controls the IEU mode integer switch 34 such that any

requests to the second subset RA[31:24] are re-directed to the shadow RT[31:24], using the same requested offset within the integer set. Any operating system kernel or other then-executing entity may thus have apparent access to the second subset RA[31:24] without the otherwise-required overhead of swapping the contents of the second subset RA[31:24] out to main memory, or pushing the second subset RA[31:24] onto a stack, or other conventional register-saving technique.

When the execution engine 14 returns to normal user mode and control passes to the originally-executing user application, the mode control unit 44 controls the IEU mode integer switch 34 such that access is again directed to the second subset RA[31:24]. In one embodiment, the mode control unit 44 is responsive to the present state of interrupt enablement in the IEU 10. In one embodiment, the execution engine 14 includes a processor status register (PSR) (not shown), which includes a one-bit flag (PSR[7]) indicating whether interrupts are enabled or disabled. Thus, the line 46 may simply couple the IEU mode integer switch 34 to the interrupts-enabled flag in the PSR. While interrupts are disabled, the IEU 10 maintains access to the integers RA[23:0], in order that it may readily perform analysis of various data of the user application. This may allow improved debugging, error reporting, or system performance analysis.

B. Register Set FB

The re-typable register set FB 20 may be thought of as including floating point registers 48 (RF[31:0]), and/or integer registers 50 (RB[31:0]). When neither data type is implied to the exclusion of the other, this application will use the term RFB[]. In one embodiment, the floating point registers RF[] occupy the same physical silicon space as the integer registers RB[]. In one embodiment, the floating point registers RF[] are sixty-four bits wide and the integer registers RB[] are thirty-two bits wide. It will be understood that if double-precision floating point numbers are not required, the register set RFB[] may advantageously be constructed in a thirty-two-bit width to save the silicon area otherwise required by the extra thirty-two bits of each floating point register.

Each individual register in the register set RFB[] may hold either a floating point value or an integer value. The register set RFB[] may include optional hardware for preventing accidental access of a floating point value as though it were an integer value, and vice versa. In one embodiment, however, in the interest of simplifying the register set RFB[], it is simply left to the software designer to ensure that no erroneous usages of individual registers are made. Thus, the execution engine 14 simply makes an access request on line 52, specifying an offset into the register set RFB[], without specifying whether the register at the given offset is intended to be used as a floating point register or an integer register. Within the execution engine 14, various entities may use either the full sixty-four bits provided by the register set RFB[], or may use only the low order thirty-two bits, such as in integer operations or single-precision floating point operations.

A first register RFB[0] 51 contains the constant value zero, in a form such that RB[0] is a thirty-two-bit integer zero (0000_{hex}) and RF[0] is a sixty-four-bit floating point zero (00000000_{hex}). This provides the same advantages as described above for RA[0].

C. Register Set C

The register set C 22 includes a plurality of Boolean registers 54 (RC[31:0]). RC[] is also known as the "condition status register" (CSR). The Boolean registers RC[] are each identical in structure and addressing, albeit that each is individually addressable at a unique address or offset within RC[].

In one embodiment, register set C further includes a "previous condition status register" (PCSR) 60, and the register set C also includes a CSR selector unit 62, which is responsive to the mode control unit 44 to select alternatively between the CSR 54 and the PCSR 60. In the one embodiment, the CSR is used when interrupts are enabled, and the PCSR is used when interrupts are disabled. The CSR and PCSR are identical in all other respects. In the one embodiment, when interrupts are set to be disabled, the CSR selector unit 62 pushes the contents of the CSR into the PCSR, overwriting the former contents of the PCSR, and when interrupts are re-enabled, the CSR selector unit 62 pops the contents of the PCSR back into the CSR. In other embodiments it may be desirable to merely alternate access between the CSR and the PCSR, as is done with RA[31:24] and RT[31:24]. In any event, the PCSR is always available as a thirty-two-bit "special register".

None of the Boolean registers is a dedicated condition flag, unlike the Boolean registers in previously known microprocessors. That is, the CSR 54 does not include a dedicated carry flag, nor a dedicated a minus flag, nor a dedicated flag indicating equality of a comparison or a zero subtraction result. Rather, any Boolean register may be the destination of the Boolean result of any Boolean operation. As with the other register sets, a first Boolean register 58 (RC[0]) always contains the value zero, to obtain the advantages explained above for RA[0]. In the preferred embodiment, each Boolean register is one bit wide, indicating one Boolean value.

II. EXECUTION ENGINE

The execution engine 14 includes one or more integer functional units 66, one or more floating point functional units 68, and one or more Boolean functional units 70. The functional units execute instructions as will be explained below. Buses 72, 73, and 75 connect the various elements of the IEU 10, and will each be understood to represent data, address, and control paths.

A. Instruction Format

FIG. 7 illustrates one exemplary format for an integer instruction which the execution engine 14 may execute. It will be understood that not all instructions need to adhere strictly to the illustrated format, and that the data processing system includes an instruction fetcher and decoder (not shown) which are adapted to operate upon varying format instructions. The single example of FIG. 7 is for ease in explanation only. Throughout this Application the identification I[] will be used to identify various bits of the instruction. I[31:30] are reserved for future implementations of the execution engine 14. I[29:26] identify the instruction class of the particular instruction. Table 1 shows the various classes of instructions performed by the present invention.

TABLE 1

	Instruction Classes
Class	Instructions
0–3	Integer and floating point register-to-register instructions

TABLE 1-continued

Instruction Classes	
Class	Instructions
4	Immediate constant load
5	Reserved
6	Load
7	Store
8–11	Control Flow
12	Modifier
13	Boolean operations
14	Reserved
15	Atomic (extended)

Instruction classes of particular interest to this Application include the Class **0-3** register-to-register instructions and the Class **13** Boolean operations. While other classes of instructions also operate upon the register file **12**, further discussion of those classes is not believed necessary in order to fully understand the present invention.

I[**25**] is identified as B**0**, and indicates whether the destination register is in register set A or register set B. I[**24:22**] are an opcode which identifies, within the given instruction class, which specific function is to be performed. For example, within the register-to-register classes, an opcode may specify "addition". I[**21**] identifies the addressing mode which is to be used when performing the instruction—either register source addressing or immediate source addressing. I[**20:16**] identify the destination register as an offset within the register set indicated by B**0**.]**2115**] is identified as B**1** and indicates whether the first operand is to be taken from register set A or register set B. I[**14:10**] identify the register offset from which the first operand is to be taken. I[**9:8**] identify a function selection—an extension of the opcode I[**24:22**]. I[**7:6**] are reserved. I[**5**] is identified as B**2** and indicates whether a second operand of the instruction is to be taken from register set A or register set B. Finally, I[**4:0**] identify the register offset from which the second operand is to be taken.

With reference to FIG. **1**, the integer functional unit **66** and floating point functional unit **68** are equipped to perform integer comparison instructions and floating point comparisons, respectively. The instruction format for the comparison instruction is substantially identical to that shown in FIG. **7**, with the caveat that various fields may advantageously be identified by slightly different names. I[**20:16**] identifies the destination register where the result is to be stored, but the addressing mode field I[**21**] does not select between register sets A or B. Rather, the addressing mode field indicates whether the second source of the comparison is found in a register or is immediate data. Because the comparison is a Boolean type instruction, the destination register is always found in register set C. All other fields function as shown in FIG. **7**. In performing Boolean operations within the integer and floating point functional units, the opcode and function select fields identify which Boolean condition is to be tested for in comparing the two operands. The integer and the floating point functional units fully support the IEEE standards for numerical comparisons.

The IEU **10** is a load/store machine. This means that when the contents of a register are stored to memory or read from memory, an address calculation must be performed in order to determine which location in memory is to be the source or the destination of the store or load, respectively. When this is the case, the destination register field I[**20:16**] identifies the register which is the destination or the source of the load or store, respectively. The source register **1** field, I[**14:10**], identifies a register in either set A or B which contains a base address of the memory location. In one embodiment, the source register **2** field, I[**4:0**], identifies a register in set A or set B which contains an index or an offset from the base. The load/store address is calculated by adding the index to the base. In another mode, I[**7:0**] include immediate data which are to be added as an index to the base.

B. Operation of the Instruction Execution Unit and Register Sets

It will be understood by those skilled in the art that the integer functional unit **66**, the floating point functional unit **68**, and the Boolean functional unit **70** are responsive to the contents of the instruction class field, the opcode field, and the function select field of a present instruction being executed.

1. Integer Operations

For example, when the instruction class, the opcode, and function select indicate that an integer register-to-register addition is to be performed, the integer functional unit may be responsive thereto to perform the indicated operation, while the floating point functional unit and the Boolean functional unit may be responsive thereto to not perform the operation. As will be understood from the cross-referenced applications, however, the floating point functional unit **68** is equipped to perform both floating point and integer operations. Also, the functional units are constructed to each perform more than one instruction simultaneously.

The integer functional unit **66** performs integer functions only. Integer operations typically involve a first source, a second source, and a destination. A given integer instruction will specify a particular operation to be performed on one or more source operands and will specify that the result of the integer operation is to be stored at a given destination. In some instructions, such as address calculations employed in load/store operations, the sources are utilized as a base and an index. The integer functional unit **66** is coupled to a first bus **72** over which the integer functional unit **66** is connected to a switching and multiplexing control (SMC) unit A **74** and an SMC unit B **76**. Each integer instruction executed by the integer functional unit **66** will specify whether each of its sources and destination reside in register set A or register set B.

Suppose that the IEU **10** has received, from the instruction fetch unit (not shown), an instruction to perform an integer register-to-register addition. In various embodiments, the instruction may specify a register bank, perhaps even a separate bank for each source and destination. In one embodiment, the instruction I[] is limited to a thirty-two-bit length, and does not contain any indication of which register bank **16-0** through **16**-n is involved in the instruction. Rather, the bank selector unit **78** controls which register bank is presently active. In one embodiment, the bank selector unit **78** is responsive to one or more bank selection bits in a status word (not shown) within the IEU **10**.

In order to perform the integer addition instruction, the integer functional unit **66** is responsive to the identification in I[**14:10**] and I[**4:0**] of the first and second source registers. The integer functional unit **66** places an identification of the first and second source registers at ports **S1** and **S2**, respectively, onto the integer functional unit bus **72** which is coupled to both SMC units A and B **74** and **76**. In one

embodiment, the SMC units A and B are each coupled to receive B0-2 from the instruction I[]. In one embodiment, a zero in any respective Bn indicates register set A, and a one indicates register set B. During load/store operations, the source ports of the integer and floating point functional units **66** and **68** are utilized as a base port and an index port, B and I, respectively.

After obtaining the first and second operands from the indicated register sets on the bus **72**, as explained below, the integer functional unit **66** performs the indicated operation upon those operands, and provides the result at port D onto the integer functional unit bus **72**. The SMC units A and B are responsive to B**0** to route the result to the appropriate register set A or B.

The SMC unit B is further responsive to the instruction class, opcode, and function selection to control whether operands are read from (or results are stored to) either a floating point register RF[] or an integer register RB[]. As indicated, in one embodiment, the registers RF[] may be sixty-four bits wide while the registers are RB[] are only thirty-two bits wide. Thus, SMC unit B controls whether a word or a double word is written to the register set RFB[]. Because all registers within register set A are thirty-two bits wide, SMC unit, A need not include means for controlling the width of data transfer on the bus **42**.

All data on the bus **42** are thirty-two bits wide, but other sorts of complexities exist within register set A. The IEU mode integer switch **34** is responsive to the mode control unit **44** of the execution engine **14** to control whether data on the bus **42** are connected through to bus **36**, bus **38** or bus **40**, and vice versa.

IEU mode integer switch **34** is further responsive to I[**20**:**16**], I[**14**:**10**], and I[**4**:**0**]. If a given indicated destination or source is in RA[**23**:**0**], the IEU mode integer switch **34** automatically couples the data between lines **42** and **36**. However, for registers RA[**31**:**24**], the IEU mode integer switch **34** determines whether data on line **42** is connected to line **38** or line **40**, and vice versa. When interrupts are enabled, IEU mode integer switch **34** connects the SMC unit A to the second subset **28** of integer registers RA[**31**:**24**]. When interrupts are disabled, the IEU mode integer switch **34** connects the SMC unit A to the shadow registers RT[**31**:**24**]. Thus, an instruction executing within the integer functional unit **66** need not be concerned with whether to address RA[**31**:**24**] or RT[**31**:**24**]. It will be understood that SMC unit A may advantageously operate identically whether it is being accessed by the integer functional unit **66** or by the floating point functional unit **68**.

2. Floating Point Operations

The floating point functional unit **68** is responsive to the class, opcode, and function select fields of the instruction, to perform floating point operations. The S**1**, S**2**, and D ports operate as described for the integer functional unit **66**. SMC unit B is responsive to retrieve floating point operands from, and to write numerical floating point results to, the floating point registers RF[] on bus **52**.

3. Boolean Operations

SMC unit C **80** is responsive to the instruction class, opcode, and function select fields of the instruction I[]. When SMC unit C detects that a comparison operation has been performed by one of the numerical functional units **66** or **68**, it writes the Boolean result over bus **56** to the Boolean register indicated at the D port of the functional unit which performed the comparison.

The Boolean functional unit **70** does not perform comparison instructions as do the integer and floating point functional units **66** and **68**. Rather, the Boolean functional unit **70** is only used in performing bitwise logical combination of Boolean register contents, according to the Boolean functions listed in Table 2.

TABLE 2

Boolean Functions

I[23,22,9,8]	Boolean result calculation
0000	ZERO
0001	S1 AND S2
0010	S1 AND (NOT S2)
0011	S1
0100	(NOT S1) AND S2
0101	S2
0110	S1 XOR S2
0111	S1 OR S2
1000	S1 NOR S2
1001	S1 XNOR S2
1010	NOT S2
1011	S1 OR (NOT S2)
1100	NOT S1
1101	(NOT S1) OR S2
1110	S1 NAND S2
1111	ONE

The advantage which the present invention obtains by having a plurality of homogenous Boolean registers, each of which is individually addressable as the destination of a Boolean operation, will be explained with reference to Tables 3–5. Table 3 illustrates an example of a segment of code which performs a conditional branch based upon a complex Boolean function. The complex Boolean function includes three portions which are OR-ed together. The first portion includes two sub-portions, which are AND-ed together.

TABLE 3

Example of Complex Boolean Function

```
1   RA[1] := 0;
2   IF (((RA[2] = RA[3]) AND (RA[4] > RA[5])) OR
3         (RA[6] < RA[7]) OR
4         (RA[8] <> RA[9])) THEN
5         X( )
6   ELSE
7         Y( );
8   RA[10] := 1;
```

Table 4 illustrates, in pseudo-assembly form, one likely method by which previous microprocessors would perform the function of Table 3. The code in Table 4 is written as though it were constructed by a compiler of at least normal intelligence operating upon the code of Table 3. That is, the compiler will recognize that the condition expressed in lines 2–4 of Table 3 is passed if any of the three portions is true.

TABLE 4

Execution of Complex Boolean Function Without Boolean Register Set

```
1   START    LDI   RA[1],0
2   TEST1    CMP   RA[2],RA[3]
3            BNE   TEST2
4            CMP   RA[4],RA[5]
5            BGT   DO_IF
```

TABLE 4-continued

Execution of Complex Boolean Function Without Boolean Register Set

6	TEST2	CMP	RA[6],RA[7]
7		BLT	DO_IF
8	TEST3	CMP	RA[8],RA[9]
9		BEQ	DO_ELSE
10	DO_IF	JSR	ADDRESS OF X( )
11		JMP	PAST_ELSE
12	DO_ELSE	JSR	ADDRESS OF Y( )
13	PAST_ELSE	LDI	RA[10],1

The assignment at line 1 of Table 3 is performed by the "load immediate" statement at line 1 of Table 4. The first portion of the complex Boolean condition, expressed at line 2 of Table 3, is represented by the statements in lines 2–5 of Table 4. To test whether RA[**2**] equals RA[**3**], the compare statement at line 2 of Table 4 performs a subtraction of RA[**2**] from RA[**3**] or vice versa, depending upon the implementation, and may or may not store the result of that subtraction. The important function performed by the comparison statement is that the zero, minus, and carry flags will be appropriately set or cleared.

The conditional branch statement at line 3 of Table 4 branches to a subsequent portion of code upon the condition that RA[**2**] did not equal RA[**3**]. If the two were unequal, the zero flag will be clear, and there is no need to perform the second sub-portion. The existence of the conditional branch statement at line 3 of Table 4 prevents the further fetching, decoding, and executing of any subsequent statement in Table 4 until the results of the comparison in line 2 are known, causing a pipeline stall. If the first sub-portion of the first portion (TEST1) is passed, the second sub-portion at line 4 of Table 4 then compares RA[**4**] to RA[**5**], again setting and clearing the appropriate status flags.

If RA[**2**] equals RA[**3**], and RA[**4**] is greater than RA[**5**], there is no need to test the remaining two portions (TEST2 and TEST3) in the complex Boolean function, and the statement at Table 4, line 5, will conditionally branch to the label DO_IF, to perform the operation inside the "IF" of Table 3. However, if the first portion of the test is failed, additional processing is required to determine which of the "IF" and "ELSE" portions should be executed.

The second portion of the Boolean function is the comparison of RA[**6**] to RA[**7**], at line 6 of Table 4, which again sets and clears the appropriate status flags. If the condition "less than" is indicated by the status flags, the complex Boolean function is passed, and execution may immediately branch to the DO_IF label. In various prior microprocessors, the "less than" condition may be tested by examining the minus flag. If RA[**7**] was not less than RA[**6**], the third portion of the test must be performed. The statement at line 8 of Table 4 compares RA[**8**] to RA[**9**]. If this comparison is failed, the "ELSE" code should be executed; otherwise, execution may simply fall through to the "IF" code at line 10 of Table 4, which is followed by an additional jump around the "ELSE" code. Each of the conditional branches in Table 4, at lines 3, 5, 7 and 9, results in a separate pipeline stall, significantly increasing the processing time required for handling this complex Boolean function.

The greatly improved throughput which results from employing the Boolean register set C of the present invention will now readily be seen with specific reference to Table 5.

TABLE 5

Execution of Complex Boolean Function With Boolean Register Set

1	START	LDI	RA[1],0
2	TEST1	CMP	RC[11],RA[2],RA[3],EQ
3		CMP	RC[12],RA[4],RA[5],GT
4	TEST2	CMP	RC[13],RA[6],RA[7],LT
5	TEST3	CMP	RC[14],RA[8],RA[9],NE
6	COMPLEX	AND	RC[15],RC[11],RC[12]
7		OR	RC[16],RC[13],RC[14]
8		OR	RC[17],RC[15],RC[16]
9		BC	RC[17],DO_ELSE
10	DO_IF	JSR	ADDRESS OF X( )
11		JMP	PAST_ELSE
12	DO_ELSE	JSR	ADDRESS OF Y( )
13	PAST_ELSE	LDI	RA[10],1

Most notably seen at lines 2–5 of Table 5, the Boolean register set C allows the microprocessor to perform the three test portions back-to-back without intervening branching. Each Boolean comparison specifies two operands, a destination, and a Boolean condition for which to test. For example, the comparison at line 2 of Table 5 compares the contents of RA[**2**] to the contents of RA[**3**], tests them for equality, and stores into RC[**11**] the Boolean value of the result of the comparison. Note that each comparison of the Boolean function stores its respective intermediate results in a separate Boolean register. As will be understood with reference to the above-referenced related applications, the IEU **10** is capable of simultaneously performing more than one of the comparisons.

After at least the first two comparisons at lines 2–3 of Table 5 have been completed, the two respective comparison results are AND-ed together as shown at line 6 of Table 3. RC[**15**] then holds the result of the first portion of the test. The results of the second and third sub-portions of the Boolean function are OR-ed together as seen in Table 5, line 7. It will be understood that, because there are no data dependencies involved, the AND at line 6 and the OR-ed in line 7 may be performed in parallel. Finally, the results of those two operations are OR-ed together as seen at line 8 of Table 5. It will be understood that register RC[**17**] will then contain a Boolean value indicating the truth or falsity of the entire complex Boolean function of Table 3. It is then possible to perform a single conditional branch, shown at line 9 of Table 5. In the mode shown in Table 5, the method branches to the "ELSE" code if Boolean register RC[**17**] is clear, indicating that the complex function was failed. The remainder of the code may be the same as it was without the Boolean register set as seen in Table 4.

The Boolean functional unit **70** is responsive to the instruction class, opcode, and function select fields as are the other functional units. Thus, it will be understood with reference to Table 5 again, that the integer and/or floating point functional units will perform the instructions in lines 1–5 and 13, and the Boolean functional unit **70** will perform the Boolean bitwise combination instructions in lines 6–8. The control flow and branching instructions in line 9–12 will be performed by elements of the IEU **10** which are not shown in FIG. **1**.

III. DATA PATHS

FIGS. **2**–**5** illustrate further details of the data paths within the floating point, integer, and Boolean portions of the IEU, respectively.

A. Floating Point Portion Data Paths

As seen in FIG. 2, the register set FB 20 is a multi-ported register set. In one embodiment, the register set FB 20 has two write ports WFB0–1, and five read ports RDFB0–4. The floating point functional unit 68 of FIG. 1 is comprised of the ALU2 102, FALU 104, MULT 106, and NULL 108 of FIG. 2. All elements of FIG. 2 except the register set 20 and the elements 102–108 comprise the SMC unit B of FIG. 1.

External, bidirectional data bus EX_DATA[] provides data to the floating point load/store unit 122. Immediate floating point data bus LDF_IMED[] provides data from a "load immediate" instruction. Other immediate floating point data are provided on busses RFF1_IMED and RFF2_IMED, such as is involved in an "add immediate" instruction. Data are also provided on bus EX_SR_DT[], in response to a "special register move" instruction. Data may also arrive from the integer portion, shown in FIG. 3, on busses 114 and 120.

The floating point register set's two write ports WFB0 and WFB1 are coupled to write multiplexers 110-0 and 110-1, respectively. The write multiplexers 110 receive data from: the ALU0 or SHF0 of the integer portion of FIG. 3; the FALU; the MULT; the ALU2; either EX_SR_DT[] or LDF_IMED[]; and EX_DATA[]. Those skilled in the art will understand that control signals (not shown) determine which input is selected at each port, and address signals (not shown) determine to which register the input data are written. Multiplexer control and register addressing are within the skill of persons in the art, and will not be discussed for any multiplexer or register set in the present invention.

The floating point register set's five read ports RDFB0 to RDFB4 are coupled to read multiplexers 112-0 to 112-4, respectively. The read multiplexers each also receives data from: either EX_SR_DT[] or LDF_IMED[], on load immediate bypass bus 126; a load external data bypass bus 127, which allows external load data to skip the register set FB; the output of the ALU2 102, which performs non-multiplication integer operations; the FALU 104, which performs non-multiplication floating point operations; the MULT 106, which performs multiplication operations; and either the ALU0 140 or the SHF0 144 of the integer portion shown in FIG. 3, which respectively perform non-multiplication integer operations and shift operations. Read multiplexers 112-1 and 112-3 also receive data from RFF1_IMED[] and RFF2_IMED[], respectively.

Each arithmetic-type unit 102–106 in the floating point portion receives two inputs, from respective sets of first and second source multiplexers S1 and S2. The first source of each unit ALU2, FALU, and MULT comes from the output of either read multiplexer 112-0 or 112-2, and the second source comes from the output of either read multiplexer 112-1 or 112-3. The sources of the FALU and the MULT may also come from the integer portion of FIG. 3 on bus 114.

The results of the ALU2, FALU, and MULT are provided back to the write multiplexers 110 for storage into the floating point registers RF[], and also to the read multiplexers 112 for re-use as operands of subsequent operations. The FALU also outputs a signal FALU_BD indicating the Boolean result of a floating point comparison operation. FALU_BD is calculated directly from internal zero and sign flags of the FALU.

Null byte tester NULL 108 performs null byte testing operations upon an operand from a first source multiplexer, in one mode that of the ALU2. NULL 108 outputs a Boolean signal NULLB_BD indicating whether the thirty-two-bit first source operand includes a byte of value zero.

The outputs of read multiplexers 112-0, 112-1, and 112-4 are provided to the integer portion (of FIG. 3) on bus 118. The output of read multiplexer 112-4 is also provided as STDT_FP[] store data to the floating point load/store unit 122.

FIG. 5 illustrates further details of the control of the S1 and S2 multiplexers. As seen, in one embodiment, each S1 multiplexer may be responsive to bit B1 of the instruction I[], and each S2 multiplexer may be responsive to bit B2 of the instruction I[]. The S1 and S2 multiplexers select the sources for the various functional units. The sources may come from either of the register files, as controlled by the B1 and B2 bits of the instruction itself. Additionally, each register file includes two read ports from which the sources may come, as controlled by hardware not shown in the Figs.

B. Integer Portion Data Paths

As seen in FIG. 3, the register set A 18 is also multi-ported. In one embodiment, the register set A 18 has two write ports WA0–1, and five read ports RDA0–4. The integer functional unit 66 of FIG. 1 is comprised of the ALU0 140, ALU1 142, SHF0 144, and NULL 146 of FIG. 3. All elements of FIG. 3 except the register set 18 and the elements 140–146 comprise the SMC unit A of FIG. 1.

External data bus EX___DATA[] provides data to the integer load/store unit 152. Immediate integer data on bus LDI_IMED[] are provided in response to a "load immediate" instruction. Other immediate integer data are provided on busses RFA1_IMED and RFA2_IMED in response to non-load immediate instructions, such as an "add immediate". Data are also provided on bus EX_SR_DT[] in response to a "special register move" instruction. Data may also arrive from the floating point portion (shown in FIG. 2) on busses 116 and 118.

The integer register set's two write ports WA0 and WA1 are coupled to write multiplexers 148-0 and 148-1, respectively. The write multiplexers 148 receive data from: the FALU or MULT of the floating point portion (of FIG. 2); the ALU0; the ALU1; the SHF0; either EX_SR_DT[] or LDI_IMED[]; and EX_DATA[].

The integer register set's five read ports RDA0 to RDA4 are coupled to read multiplexers 150-0 to 150-4, respectively. Each read multiplexer also receives data from: either EX_SR_DT[] or LDI_IMED[] on load immediate bypass bus 160; a load external data bypass bus 154, which allows external load data to skip the register set A; ALU0; ALU1; SHF0; and either the FALU or the MULT of the floating point portion (of FIG. 2). Read multiplexers 150-1 and 150-3 also receive data from RFA1_IMED[] and RFA2_IMED[], respectively.

Each arithmetic-type unit 140–144 in the integer portion receives two inputs, from respective sets of first and second source multiplexers S1 and S2. The first source of ALU0 comes from either the output of read multiplexer 150-2, or a thirty-two-bit wide constant zero (0000_{hex}), or floating point read multiplexer 112-4. The second source of ALU0 comes from either read multiplexer 150-3 or floating point read multiplexer 112-1. The first source of ALU1 comes from either read multiplexer 150-0 or IF_PC[]. IF_PC[] is used in calculating a return address needed by the instruction fetch unit (not shown), due to the IEU's ability to perform instructions in an out-of-order sequence. The second source of ALU1 comes from either read multiplexer 150-1 or CF_OFFSET[]. CF_OFFSET[] is used in calculating a return address for a CALL instruction, also due to the out-of-order capability.

The first source of the shifter SHF0 144 is from either: floating point read multiplexer 112-0 or 112-4; or any integer read multiplexer 150. The second source of SHF0 is from either: floating point read multiplexer 112-0 or 112-4; or integer read multiplexer 150-0, 150-2, or 150-4. SHF0 takes a third input from a shift amount multiplexer (SA). The third input controls how far to shift, and is taken by the SA multiplexer from either: floating point read multiplexer 112-1; integer read multiplexer 150-1 or 150-3; or a five-bit wide constant thirty-one (11111_2 or 31_{10}). The shifter SHF0 requires a fourth input from the size multiplexer (S). The fourth input controls how much data to shift, and is taken by the S multiplexer from either: read multiplexer 150-1; read multiplexer 150-3; or a five-bit wide constant sixteen (10000_2 or 16_{10}).

The results of the ALU0, ALU1, and SHF0 are provided back to the write multiplexers 148 for storage into the integer registers RA[], and also to the read multiplexers 150 for re-use as operands of subsequent operations. The output of either ALU0 or SHF0 is provided on bus 120 to the floating point portion of FIG. 3. The ALU0 and ALU1 also output signals ALU0_BD and ALU1_BD, respectively, indicating the Boolean results of integer comparison operations. ALU0_BD and ALU1_BD are calculated directly from the zero and sign flags of the respective functional units. ALU0 also outputs signals EX_TADR[] and EX_VM_ADR. EX_TADR[] is the target address generated for an absolute branch instruction, and is sent to the IFU (not shown) for fetching the target instruction. EX_VM_ADR[] is the virtual address used for all loads from memory and stores to memory, and is sent to the VMU (not shown) for address translation.

Null byte tester NULL 146 performs null byte testing operations upon an operand from a first source multiplexer. In one embodiment, the operand is from the ALU0. NULL 146 outputs a Boolean signal NULLA_BD indicating whether the thirty-two-bit first source operand includes a byte of value zero.

The outputs of read multiplexers 150-0 and 150-1 are provided to the floating point portion (of FIG. 2) on bus 114. The output of read multiplexer 150-4 is also provided as STDT_INT[] store data to the integer load/store unit 152.

A control bit PSR[7] is provided to the register set A 18. It is this signal which, in FIG. 1, is provided from the mode control unit 44 to the IEU mode integer switch 34 on line 46. The IEU mode integer switch is internal to the register set A 18 as shown in FIG. 3.

FIG. 6 illustrates further details of the control of the S1 and S2 multiplexers. The signal ALU0_BD

C. Boolean Portion Data Paths

As seen in FIG. 4, the register set C 22 is also multiported. In one embodiment, the register set C 22 has two write ports WC0–1, and five read ports RDA0–4. All elements of FIG. 4 except the register set 22 and the Boolean combinational unit 70 comprise the SMC unit C of FIG. 1.

The Boolean register set's two write ports WC0 and WC1 are coupled to write multiplexers 170-0 and 170-1, respectively. The write multiplexers 170 receive data from: the output of the Boolean combinational unit 70, indicating the Boolean result of a Boolean combinational operation; ALU0_BD from the integer portion of FIG. 3, indicating the Boolean result of an integer comparison; FALU_BD from the floating point portion of FIG. 2, indicating the Boolean result of a floating point comparison; either ALU1_BD_P from ALU1, indicating the results of a compare instruction in ALU1, or NULLA_BD from NULL 146, indicating a null byte in the integer portion; and either ALU2_BD_P from ALU2, indicating the results of a compare operation in ALU2, or NULLB_BD from NULL 108, indicating a null byte in the floating point portion. In one mode, the ALU0_BD, ALU1_BD, ALU2_BD, and FALU_BD signals are not taken from the data paths, but are calculated as a function of the zero flag, minus flag, carry flag, and other condition flags in the PSR. In one mode, wherein up to eight instructions may be executing at one instant in the IEU, the IEU maintains up to eight PSRs.

The Boolean register set C is also coupled to bus EX_SR_DT[], for use with "special register move" instructions. The CSR may be written or read as a whole, as though it were a single thirty-two-bit register. This enables rapid saving and restoration of machine state information, such as may be necessary upon certain drastic system errors or upon certain forms of grand scale context switching.

The Boolean register set's five read ports RDC0 to RDC3 are coupled to read multiplexers 172-0 to 172-4, respectively. The read multiplexers 172 receive the same set of inputs as the write multiplexers 170 receive. The Boolean combinational unit 70 receives inputs from read multiplexers 170-0 and 170-1. Read multiplexers 172-2 and 172-3 respectively provide signals BLBP_CPORT and BLBP_DPORT. BLBP_CPORT is used as the basis for conditional branching instructions in the IEU. BLBP_DPORT is used in the "add with Boolean" instruction, which sets an integer register in the A or B set to zero or one (with leading zeroes), depending upon the content of a register in the C set. Read port RDC4 is presently unused, and is reserved for future enhancements of the Boolean functionality of the IEU.

IV. CONCLUSION

While the features and advantages of the present invention have been described with respect to particular embodiments thereof, and in varying degrees of detail, it will be appreciated that the invention is not limited to the described embodiments. The following claims define the invention to be afforded patent coverage.

We claim:

1. In a data processing system, which includes a central processing unit (CPU) which performs operations according to an instruction, the operations operating upon integer data, a data register system comprising:

a first register set including a plurality of first registers each for holding the integer data;

a second register set including a plurality of second registers each for holding the integer data and for holding floating point data, wherein the instruction includes a field specifying which of the first and second register sets is to be accessed in response to the instruction;

means, responsive to the field, for accessing the first register set or the second register set as specified by the field, including

i) reading means for reading an operand value from either the first register set or second register set as specified by the field, and

ii) writing mean for writing a result value to the first register set or the second register set as specified by the field.

2. The apparatus of claim 1, wherein said first and second register sets each have two write ports and five read ports.

3. An apparatus comprising:

execution means for executing instructions, the instructions performing operations upon operands to generate results, each instruction specifying a respective source address for each operand and a destination address for the result of the instruction, each address specifying a register set and an offset;

a first register set including a plurality of individually addressable registers each for storing integer values;

first access means for writing and reading values to and from the first register set according to a given instruction, the first access means including,

i) first reading means, responsive to the given instruction having a given source address which specifies the first register set as a source for an operand of the given instruction, for reading the operand's value from the first register set at the offset specified by the given source address, and

ii) first writing means, responsive to the given instruction having a given destination address which specifies the first register set as a destination for the result of the given instruction, for writing the result's value to the first register set at the offset specified by the given destination address;

a second register set including a plurality of individually addressable registers, wherein each addressable register is configured to store floating point values and integer values; and

second access means for writing and reading integer and floating point values to and from the second register set according to the given instruction, the second access means including,

i) second reading means, responsive to the given instruction having a given source address which specifies the second register set as a source for an operand of the given instruction, for reading the operand's value from the second register set at the offset specified by the given source address, and

ii) second writing means, responsive to the given instruction having a given destination address which specifies the second register set as a destination for the result of the given instruction, for writing the result's value to the second register set at the offset specified by the given destination address.

4. The apparatus of claim 3, wherein:

a given instruction may specify a first and a second source address and a destination address, with each address specifying either of the first or second register sets such that the given instruction requires access to both register sets; and

the first and second access means operate simultaneously to provide the instruction parallel access to both the first and second register sets.

5. The apparatus of claim 2, wherein said first and second register sets each have two write ports and five read ports.

* * * * *